2025 Predictions for AI Agents

"By this time next year, you'll have a team of agents working for you." - Charles Lamanna, Corporate Vice President, Microsoft

Melanie Mitchell, a professor at the Santa Fe Institute, warns tha ts' mistakes could have "big consequences," particularly if they have access to personal or financial informatic

"In 2025, we'll begin to see a shift from chatbots and image gene ntic" systems that can act autonomously to cor

"The capabilities of A lf will begin to mature. This will crea to be part of hybrid teams of humans an

"Agents are the new apps"

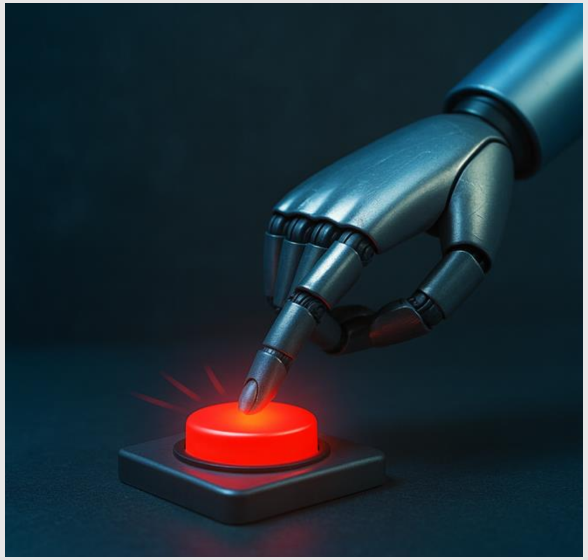
- Dharmesh Shah (Hubspot CTO and co-founder – September 2024)

Jaime Sevilla, director of AI forecasting nonprofit Epoch AI, envisions a future where AI agents function as virtual co-workers, but says that in 2025 AI agents will be mostly about their novelty.

"IBM and Morning Consult did a survey of 1,000 developers who are building AI applications for enterprise, and 99% of them said they are exploring or developing AI agents. So yes, the answer is that 2025 is going to be the year of the agent." - Maryam Ashoori, IBM Watsonx.ai

Understanding and Working with AI Agents

(A Hands-on Gen AI Workshop)



Presented by Brent Laster

bandwidth

TECHUPSKILLS ®

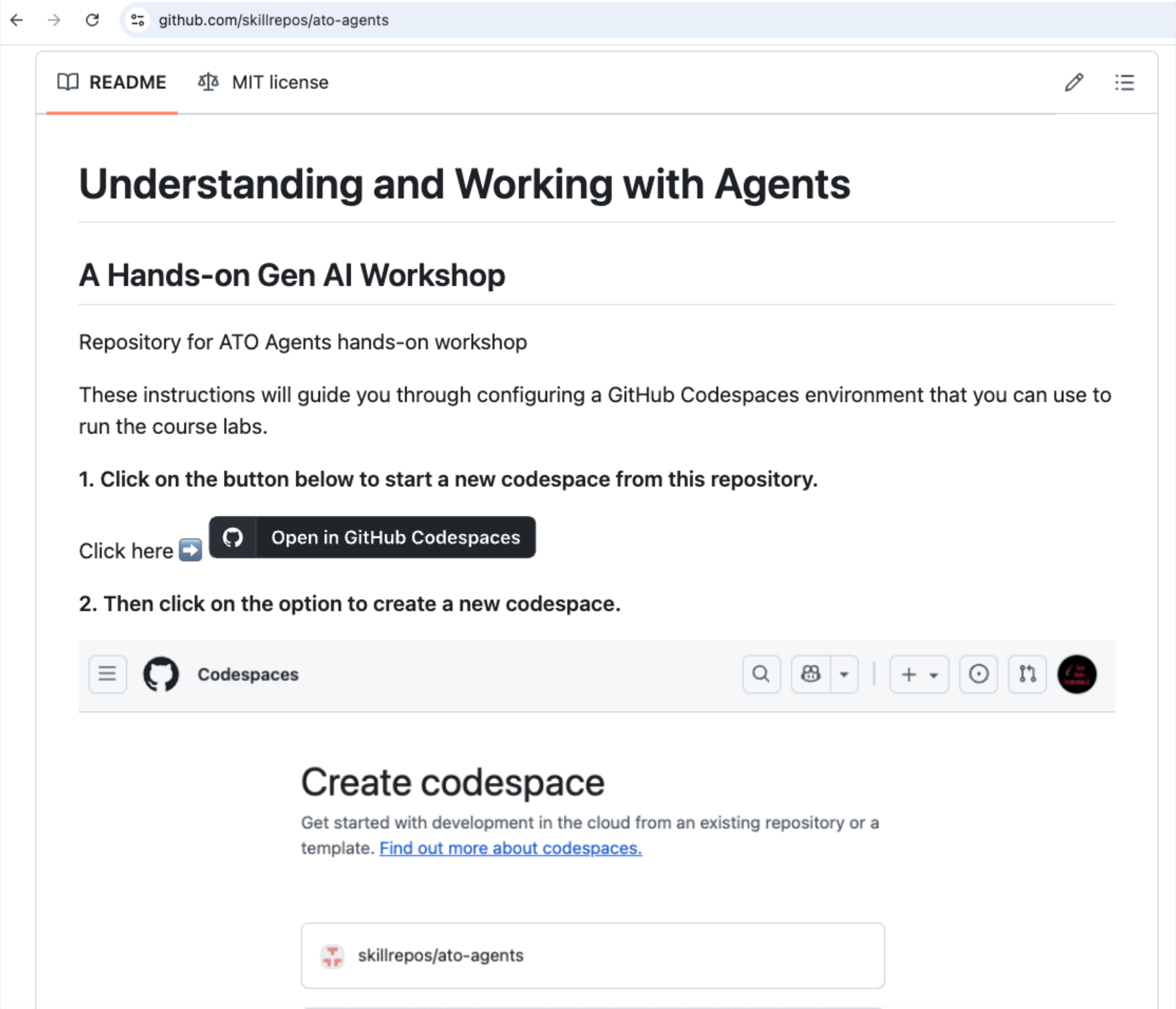
Tech Skills Transformations LLC

Agenda

- What are AI agents and how do they work?
- Agency, Chain of Thought and other important concepts
- Frameworks
- Use of memory in agents
- Coding agents
- What is Retrieval Augmented Generation (RAG)?
- Using RAG with agents
- Multi-agent structures
- Agent design patterns
- Tips on building good agents
- Challenges, concerns, strategies
- Future of AI agents

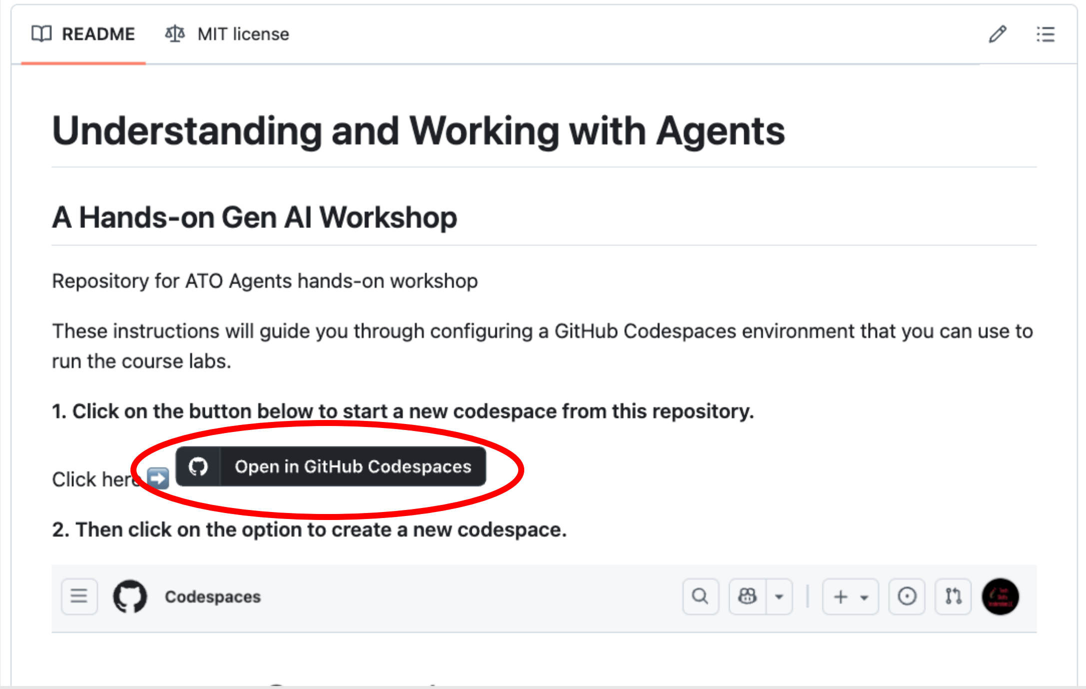
AI Agents Workshop - prereqs

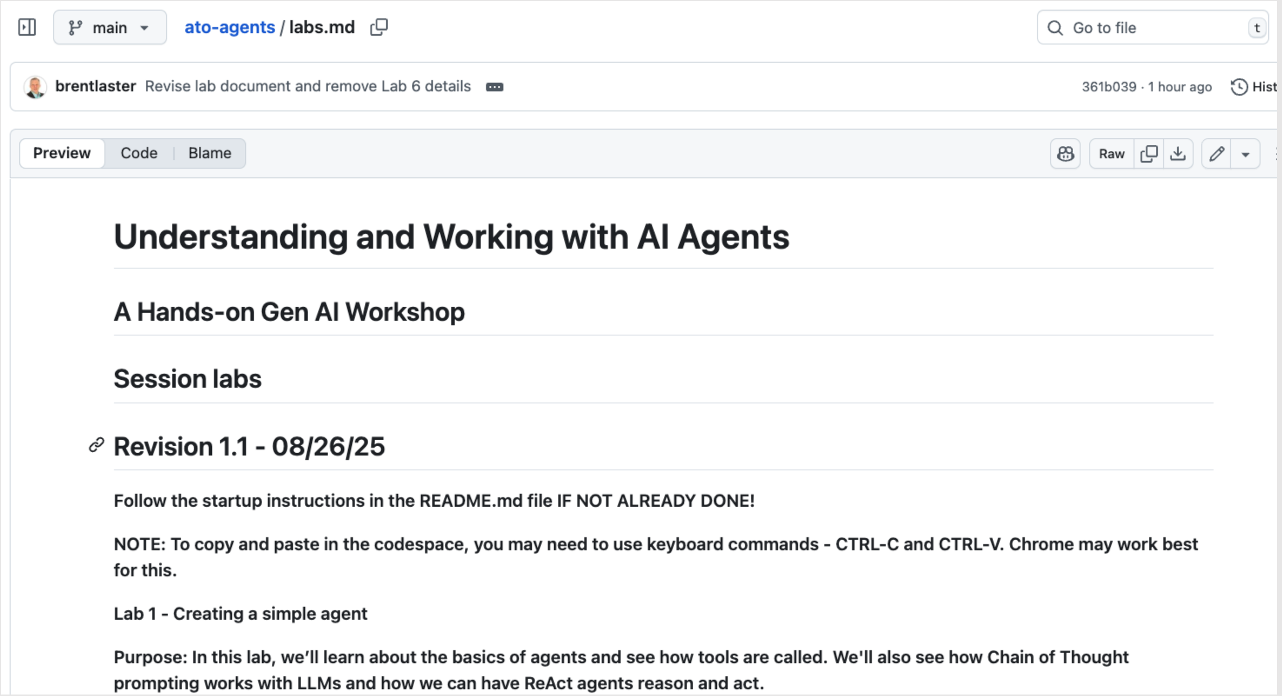
- Access to public GitHub.com
- GitHub repo for workshop is https://github.com/skillrepos/ato-agents
- For labs environment, recommended to use premade Codespace to run in (more on that in a moment) - follow instructions in README.md
- Could set up own environment (see scripts directory) but labs are geared to codespace setup



Lab prep - repo is github.com/skillrepos/ato-agents

1. Go to https://github.com/skillrepos/ato-agents (Chrome may work best for copy and paste actions.)
2. Follow instructions in **README.md**
3. Startup codespace with quickstart button in README.
4. When codespace is ready, run **scripts/setup.sh** to complete setup.



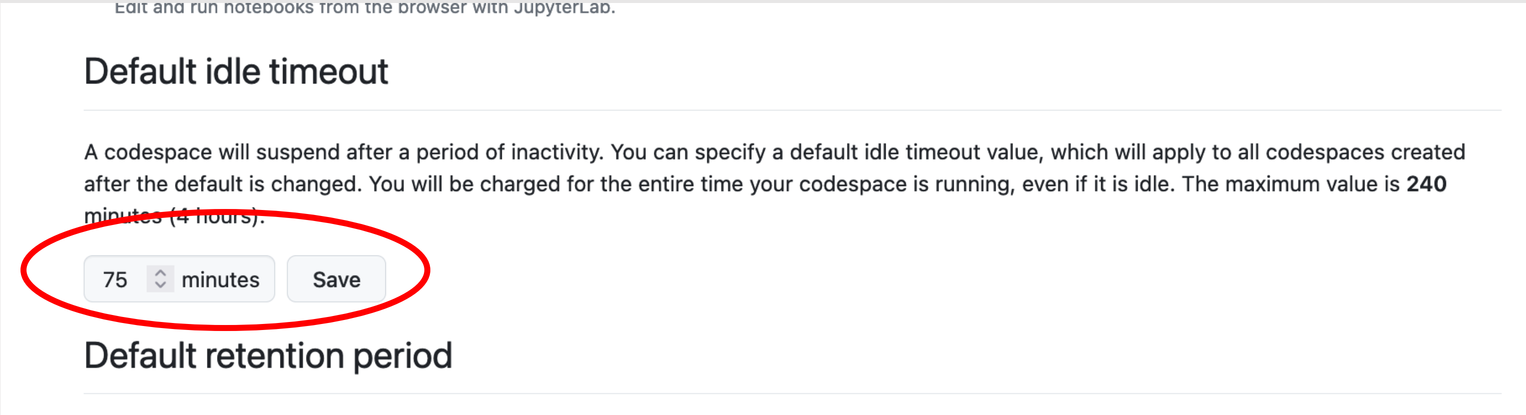


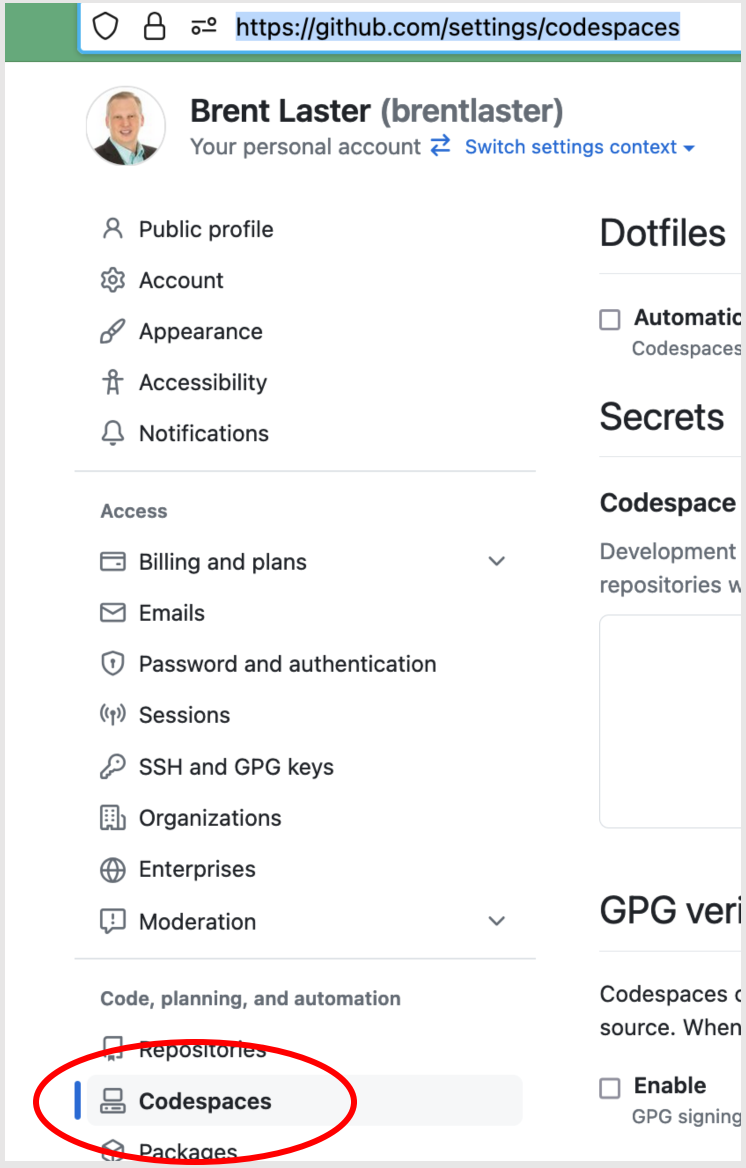
Logistics

- Workshop is lecture + labs
- Will have 15 minute intermission
- Breaks to do labs+

Codespace timeouts

- May want to set timeouts for longer than default
- When logged into GitHub, go to https://github.com/settings/codespaces
- Scroll down to find Default



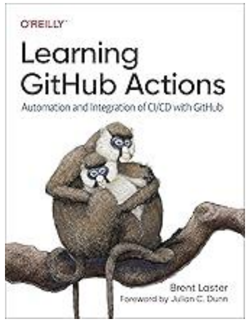


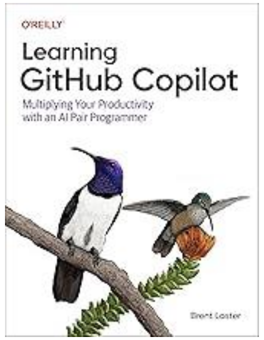
About me

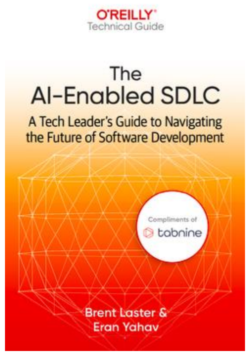
- LinkedIn: brentlaster
- X: @BrentCLaster
- Bluesky: brentclaster.bsky.social
- GitHub: brentlaster

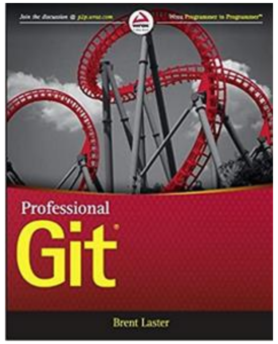
Long career in corporate:

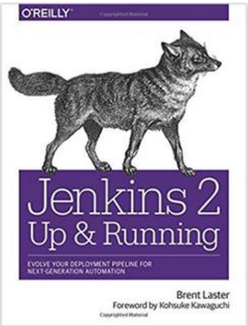
- *Principal Dev*
- *Manager/Senior Manager*
- *Director*











- Founder, Tech Skills Transformations LLC
- https://getskillsnow.com
- info@getskillsnow.com

With Tech Skills Transformations, you don't have to imagine. With new AI training on agents, MCP, RAG, LLMs, and traditional DevOps training from Git to Kubernetes, we provide the understanding, skill development, and productivity you've been looking for. Every workshop incorporates hands-on experiences to help you build confidence, proficiency, while learning how the tech works and applies to you. At Tech Skills Transformations, your success, understanding, and growth is our goal.

Connect: LinkedIn • Web: getskillsnow.com • Email: info@getskillsnow.com

Learn More »

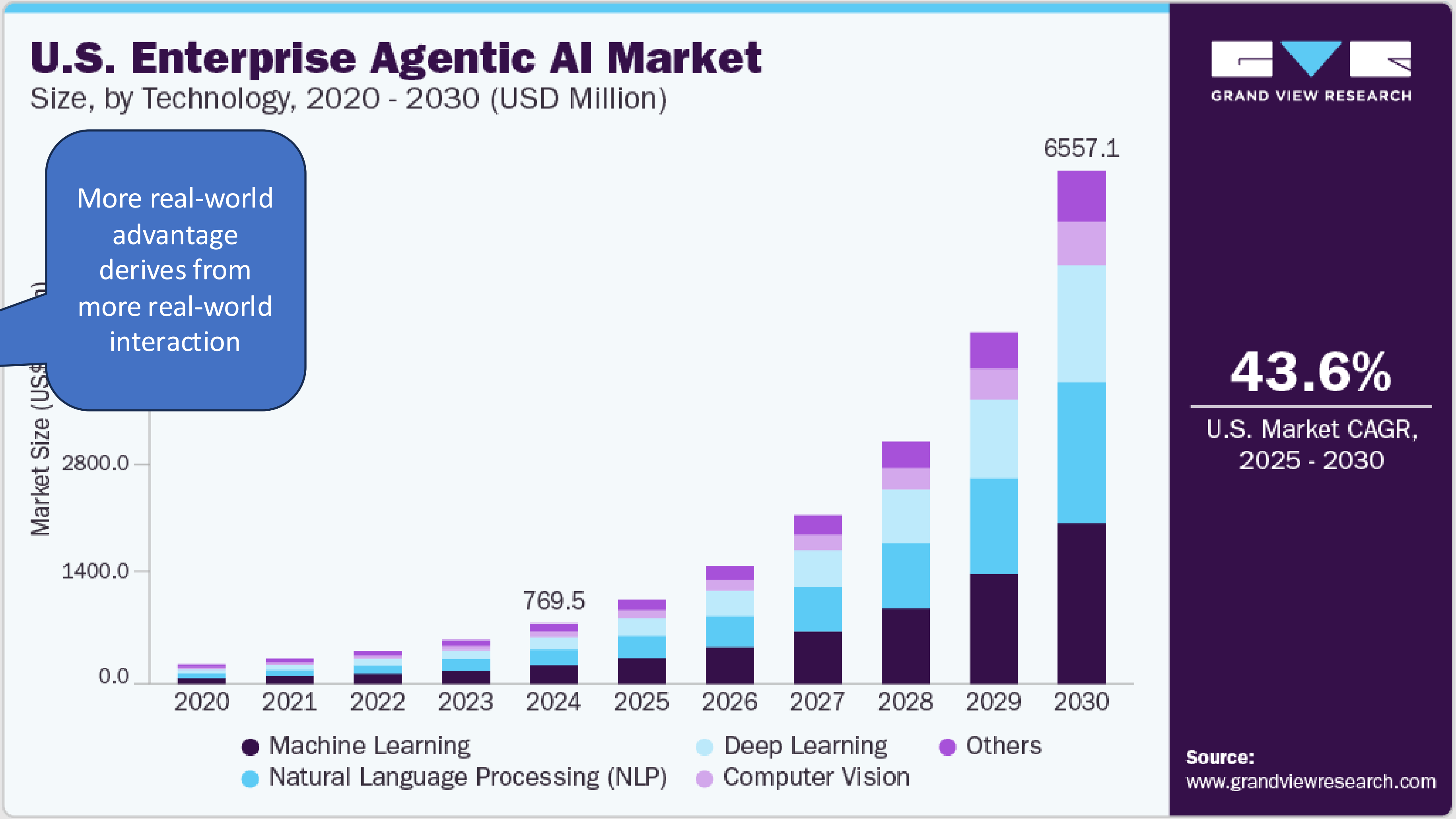
Agents - why now?

- *Converging Forces*

- AI market growth
- Smarter LLMs – from tiny to > 1T parameters
- Cheap, composable tool APIs
- Business demand
  - » autonomous automation
  - » rapid decision making
  - » competitive advantage

- Key takeaways:
  - » *AI is becoming a commodity*
  - » *Agents allow it to perceive and affect the real world*

More real-world advantage derives from more real-world interaction



CAGR = Compounded Annual Growth Rate

How can agents benefit us in the real world?

Increased Productivity

Automate routine tasks
Reduce human effort
Boost efficiency

Data-driven Insights

Large dataset analysis
Generate recommendations for decision-making

Personalized Assistance

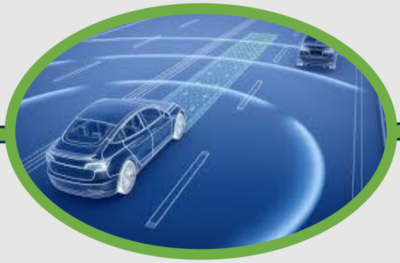
Improve user interactions
Via chatbots & virtual assts

Industry Applications

Power advanced solutions in finance, transportation, healthcare, etc.

AI Agent

Transportation



Advanced driver-assist systems

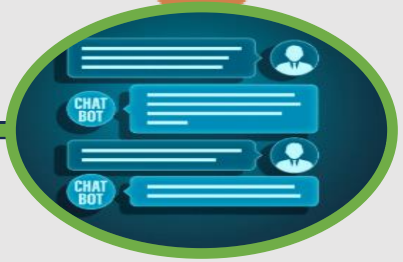
Healthcare

Health records management and diagnosis

Finance

Customer Relationship Management

Customer Service



Sentiment analysis chatbot

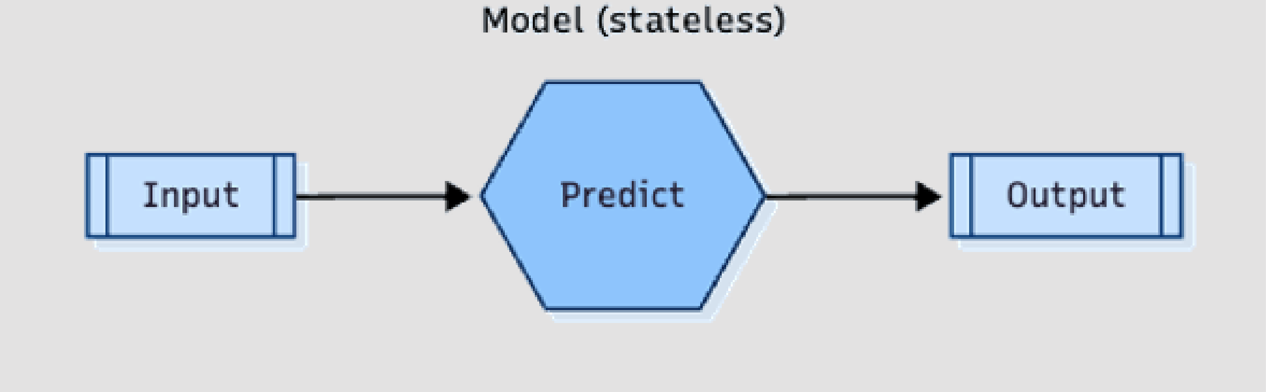
Can Agents Make a Difference?

Organization	Use Case	Key Positive Findings	Noted Negatives	Timeframe
Lenovo [1]	Customer Service & Content Generation	- 8x faster content creation - 50% faster customer response - 80% legal productivity boost	Potential over-reliance on automation	Early–mid 2025
Mayo Clinic [2]	Healthcare Diagnostics	- 89% diagnostic accuracy - 60% time reduction - 120+ FDA-approved AI devices	Concerns over algorithmic bias, need for human review	First half 2025
JPMorgan Chase [3]	Financial Trading & Agreements	- 75% equity trades by agent - 50,000+ contracts auto-processed - Quantum security	Vigilant compliance/risk monitoring required	By mid-2025

1. Lenovo AI agent study on content generation and legal productivity, including efficiency improvements and caution on automation reliance.
2. Mayo Clinic healthcare AI diagnostics study demonstrating accuracy and efficiency gains, with notes on bias and human oversight needs.
3. JPMorgan Chase AI agent impact on trading and contract processing, highlighting adoption scale and compliance concerns.

So what is an AI agent?

- Agent ≠ Model
- Uses LLMs for reasoning and communication
- Observes→ Thinks → Acts (autonomously)



Ag

system_message=“””You are an AI assistant designed to help users find weather conditions. Your primary goal is to provide precise, helpful, and clear responses.

You have access to the following tools:

Tool Name: find_weather, Description: Get weather for a location., Arguments: latitude: float, longitude: float, Outputs: string

You should think step by step in order to fulfill the objective with a reasoning process divided into Thought/Action/Observation. This cycle can repeat multiple times if needed.

You should first reflect with “Thought: {your_thoughts}” on the current query, then (if necessary), call a tool with the proper JSON formatting “Action: {JSON_BLOB}”, or else print your final answer starting with the prefix “Final Answer:”“””

Agent Example

system_message="""You are an AI assistant designed to help users find weather conditions. Your primary goal is to provide precise, helpful, and clear responses.
You have access to the following tools:
Tool Name: find_weather, Description: Get weather for a location., Arguments: latitude: float, longitude: float, Outputs: string
You should think step by step in order to fulfill the objective with a reasoning process divided into Thought/Action/Observation. This cycle can repeat multiple times if needed.
You should first reflect with "Thought: {your_thoughts}" on the current query, then (if necessary), call a tool with the proper JSON formatting "Action: {JSON_BLOB}", or else print your final answer starting with the prefix "Final Answer:" """

What's the weather in Paris?

User

LLM

Chain of Thought – Step 1: Interpret User Query
Thought: "The user is asking about the weather in Paris. I need to extract 'Paris' as the location.
Action: Extracted location = "Paris"

Weather Search Tool

AI Agent

Agent Example

system_message="""You are an AI assistant designed to help users find weather conditions. Your primary goal is to provide precise, helpful, and clear responses.
You have access to the following tools:
Tool Name: find_weather, Description: Get weather for a location., Arguments: latitude: float, longitude: float, Outputs: string
You should think step by step in order to fulfill the objective with a reasoning process divided into Thought/Action/Observation. This cycle can repeat multiple times if needed.
You should first reflect with "Thought: {your_thoughts}" on the current query, then (if necessary), call a tool with the proper JSON formatting "Action: {JSON_BLOB}", or else print your final answer starting with the prefix "Final Answer:" """

What's the weather in Paris?

User

LLM

```
AIResponse(
     tool_calls=[{
          name:
"find_weather"
          parameters:  {
               latitude:
"48.8566",
               longitude:
"2.3522",
          },
          id:  "call_tool123",
          type:  "tool_invoke"
     }]
)
```

Chain of Thought – Step 2: Decide to use tool
Thought: "I need real-time data, so I will call the '**find_weather**' tool. First, I need to get the latitude and longitude for the tool call.

Weather Search Tool

AI Agent

Agent Example

system_message="""You are an AI assistant designed to help users find weather conditions. Your primary goal is to provide precise, helpful, and clear responses.
You have access to the following tools:
Tool Name: find_weather, Description: Get weather for a location., Arguments: latitude: float, longitude: float, Outputs: string
You should think step by step in order to fulfill the objective with a reasoning process divided into Thought/Action/Observation. This cycle can repeat multiple times if needed.
You should first reflect with "Thought: {your_thoughts}" on the current query, then (if necessary), call a tool with the proper JSON formatting "Action: {JSON_BLOB}", or else print your final answer starting with the prefix "Final Answer:""""

What's the weather in Paris?

User

LLM

```
AIResponse(
    tool_calls=[{
        name: "find_weather"
        parameters: {
            latitude: "48.8566",
            longitude: "2.3522",
        },
        id: "call_tool123",
        type: "tool_invoke"
    }]
)
```

Agent parses LLM output identifies JSON tool call, parses it, forms it into actual tool call

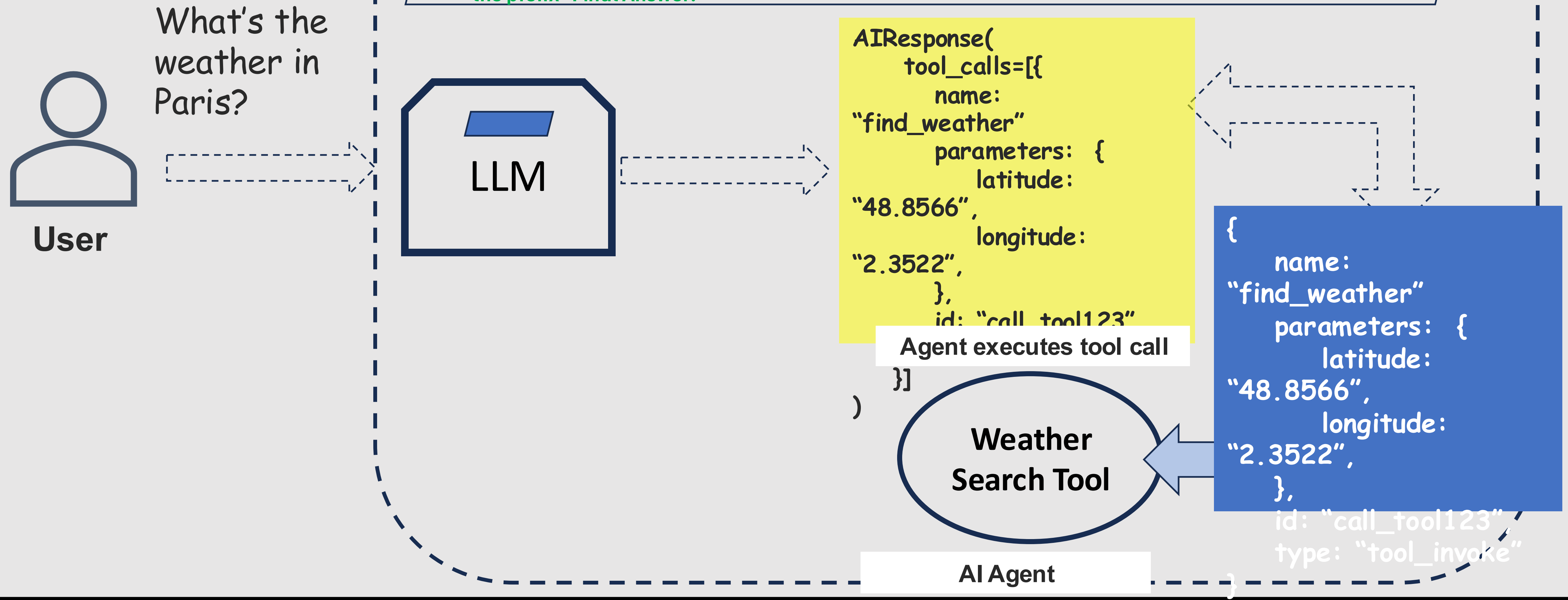
```
{
    name: "find_weather"
    parameters: {
        latitude: "48.8566",
        longitude: "2.3522",
    },
    id: "call_tool123",
    type: "tool_invoke"
}
```

Weather Search Tool

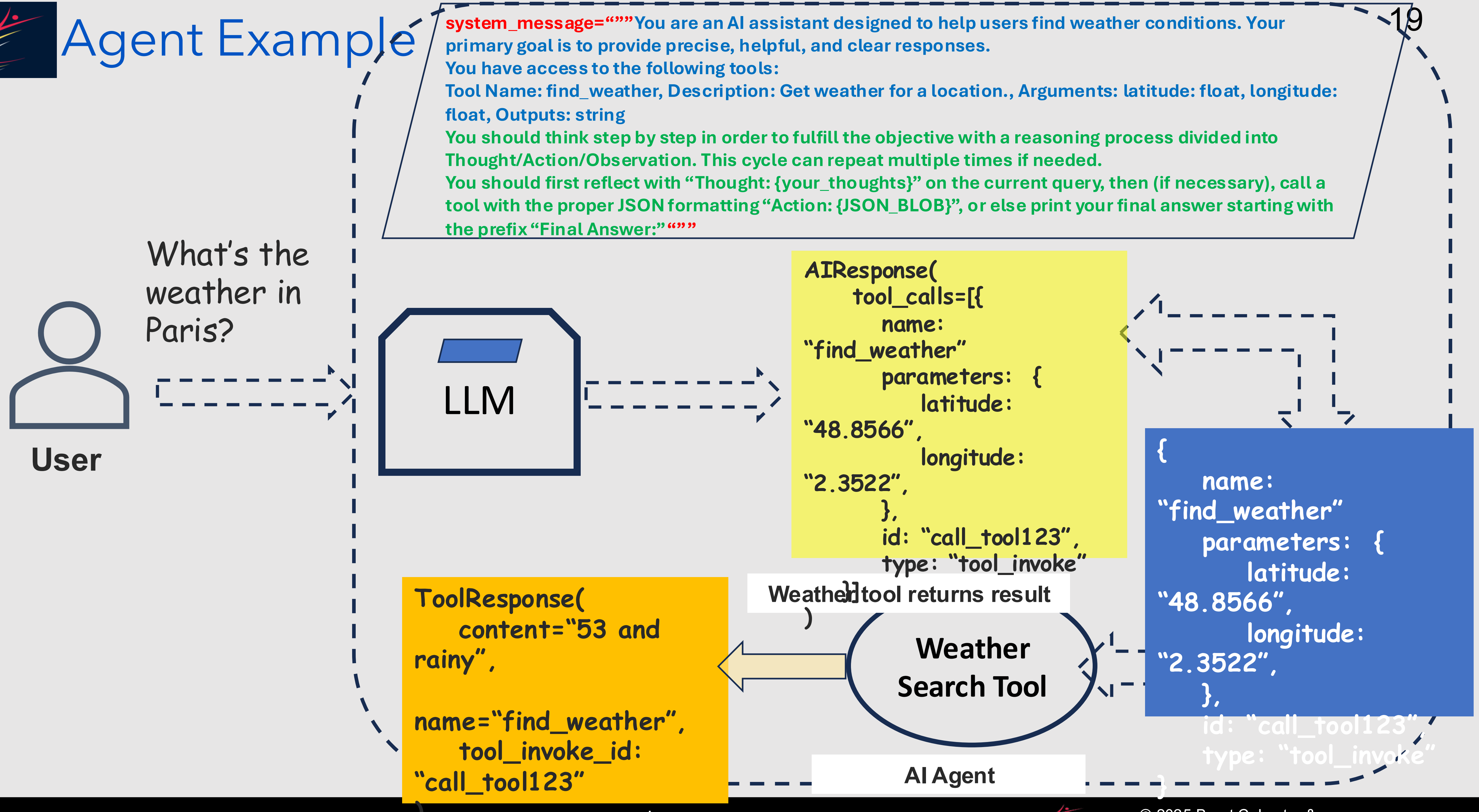
AI Agent

Agent Example

system_message="""You are an AI assistant designed to help users find weather conditions. Your primary goal is to provide precise, helpful, and clear responses.
You have access to the following tools:
Tool Name: find_weather, Description: Get weather for a location., Arguments: latitude: float, longitude: float, Outputs: string
You should think step by step in order to fulfill the objective with a reasoning process divided into Thought/Action/Observation. This cycle can repeat multiple times if needed.
You should first reflect with "Thought: {your_thoughts}" on the current query, then (if necessary), call a tool with the proper JSON formatting "Action: {JSON_BLOB}", or else print your final answer starting with the prefix "Final Answer:""""

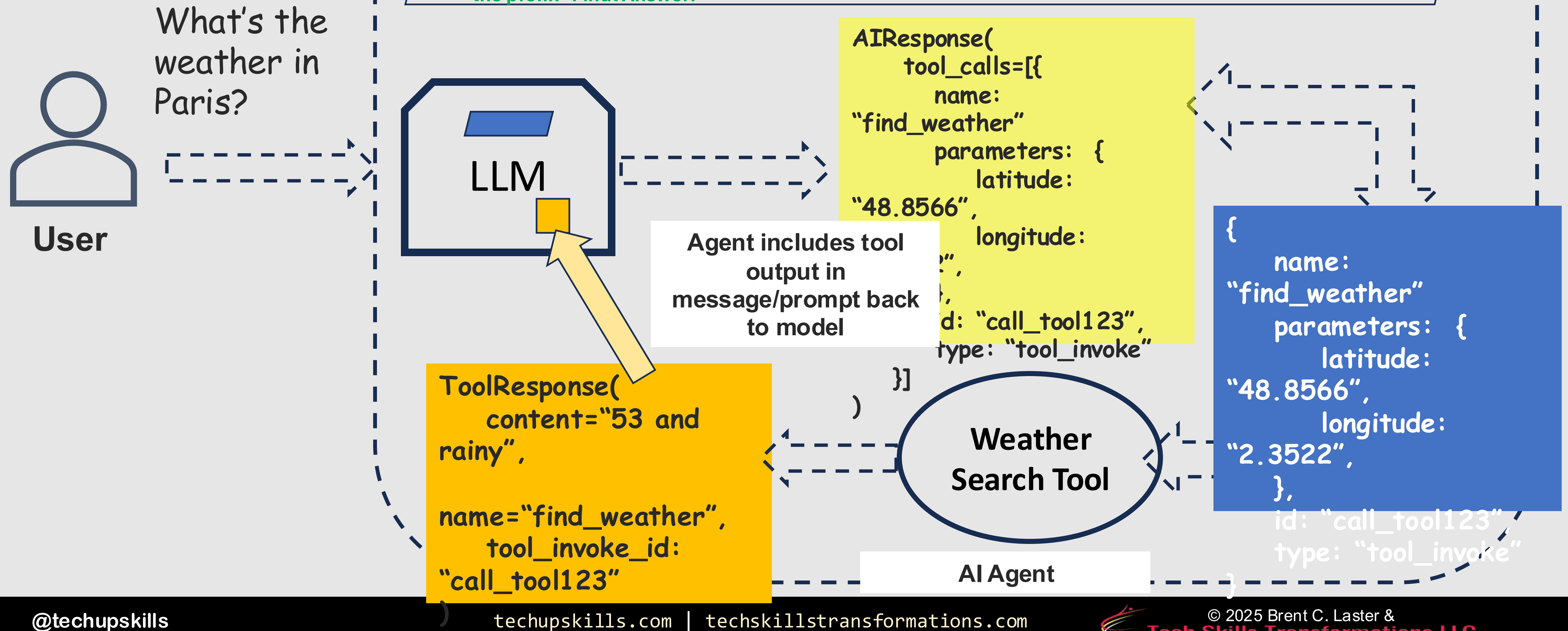


Agent Example

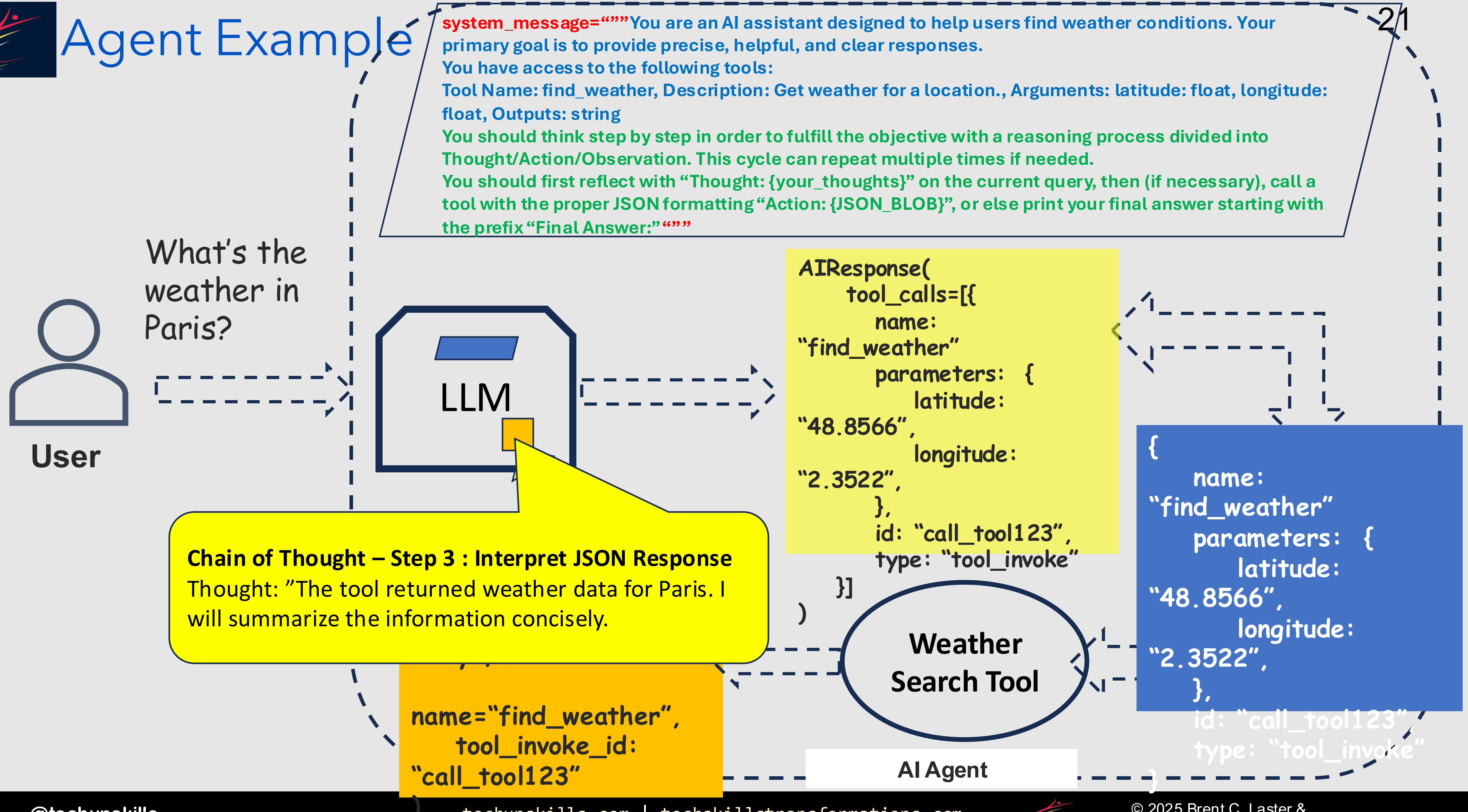


Agent Example

system_message="""You are an AI assistant designed to help users find weather conditions. Your primary goal is to provide precise, helpful, and clear responses.
You have access to the following tools:
Tool Name: find_weather, Description: Get weather for a location., Arguments: latitude: float, longitude: float, Outputs: string
You should think step by step in order to fulfill the objective with a reasoning process divided into Thought/Action/Observation. This cycle can repeat multiple times if needed.
You should first reflect with "Thought: {your_thoughts}" on the current query, then (if necessary), call a tool with the proper JSON formatting "Action: {JSON_BLOB}", or else print your final answer starting with the prefix "Final Answer:" """

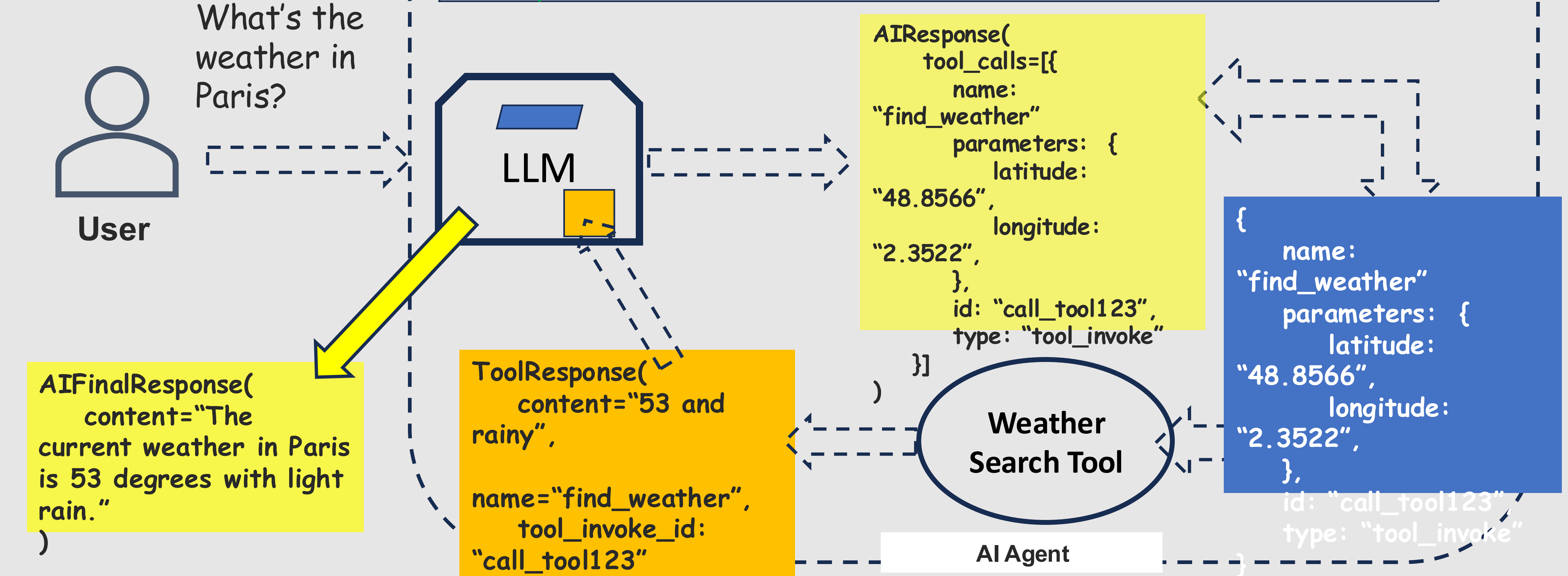


Agent Example



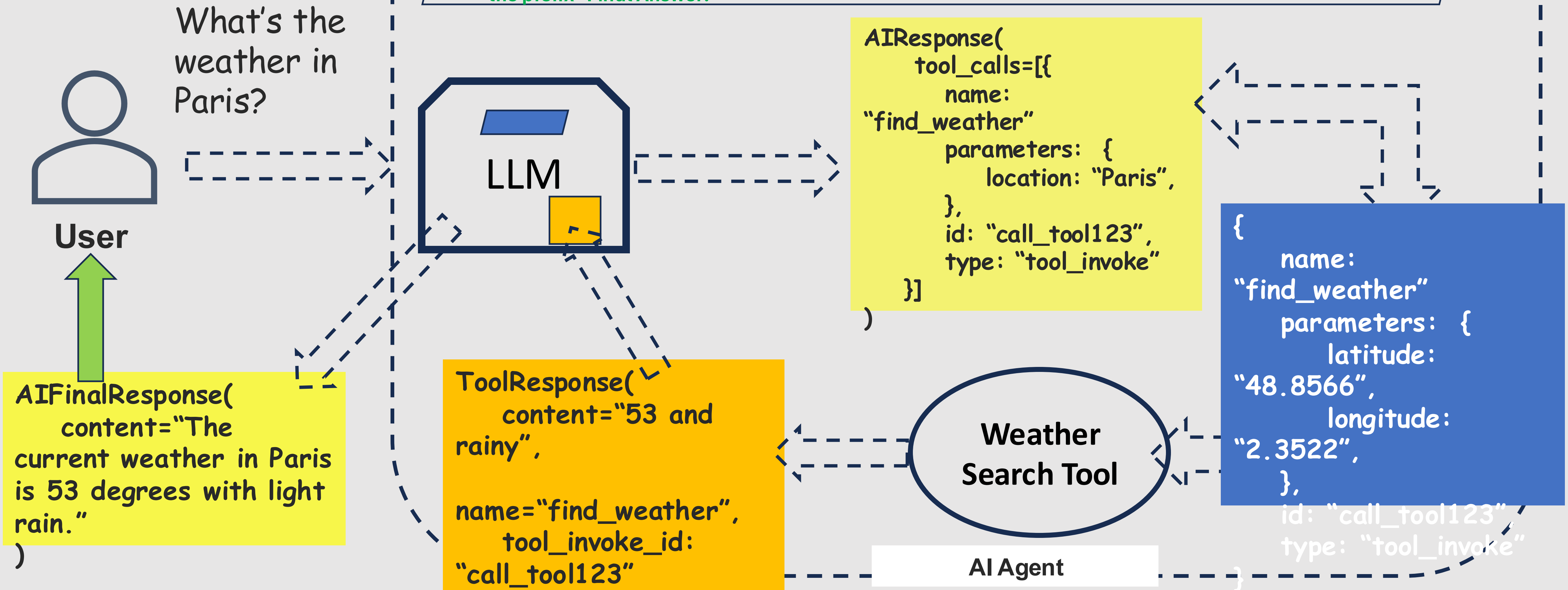
Agent Example

system_message="""You are an AI assistant designed to help users find weather conditions. Your primary goal is to provide precise, helpful, and clear responses.
You have access to the following tools:
Tool Name: find_weather, Description: Get weather for a location., Arguments: latitude: float, longitude: float, Outputs: string
You should think step by step in order to fulfill the objective with a reasoning process divided into Thought/Action/Observation. This cycle can repeat multiple times if needed.
You should first reflect with "Thought: {your_thoughts}" on the current query, then (if necessary), call a tool with the proper JSON formatting "Action: {JSON_BLOB}", or else print your final answer starting with the prefix "Final Answer:" """"



Agent Example

system_message="""You are an AI assistant designed to help users find weather conditions. Your primary goal is to provide precise, helpful, and clear responses.
You have access to the following tools:
Tool Name: find_weather, Description: Get weather for a location., Arguments: latitude: float, longitude: float, Outputs: string
You should think step by step in order to fulfill the objective with a reasoning process divided into Thought/Action/Observation. This cycle can repeat multiple times if needed.
You should first reflect with "Thought: {your_thoughts}" on the current query, then (if necessary), call a tool with the proper JSON formatting "Action: {JSON_BLOB}", or else print your final answer starting with the prefix "Final Answer:""""



Demo – LLMs and Agency

Purpose: Looking at the (lack of) agency in LLMs

Where do agents fit in the AI spectrum?

Low Agency
Static
Reactive
Simple tasks
Simple environment
Supervised

GPT-5

Claude

Modern LLMs

Gemini

alexa

AI Assistant

AI Copilot

AI AGENT

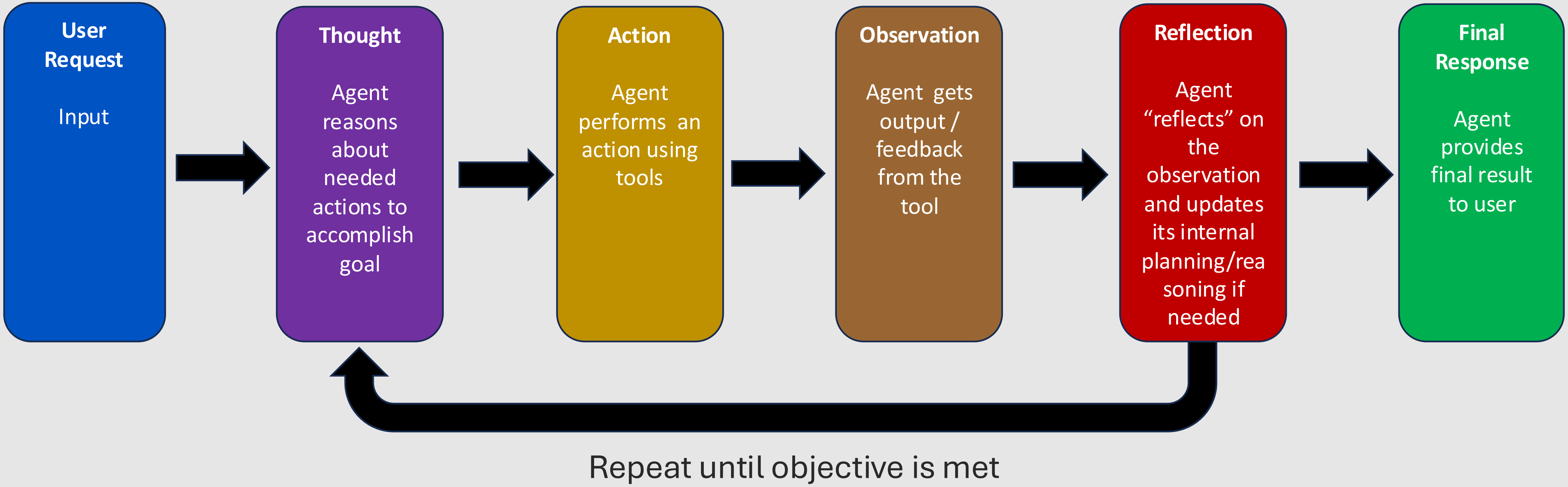
AI Agent

High Agency
Adaptive
Proactive planning
Complex goals
Complex environment
Autonomous

Level of Agency

How Agents Process Tasks – One Approach: Thought -> Action -> Observation cycle (aka **ReAct** – Reasoning and Acting framework)

- Thought: Agent "thinks" about task. Figures out what needs to be done next
- Action: Once it knows what to do, takes action (i.e. asking for info, invoking API, etc.)
- Observation: After taking action, agent looks at and evaluates (reflects on) results
- Repeats until task is completed



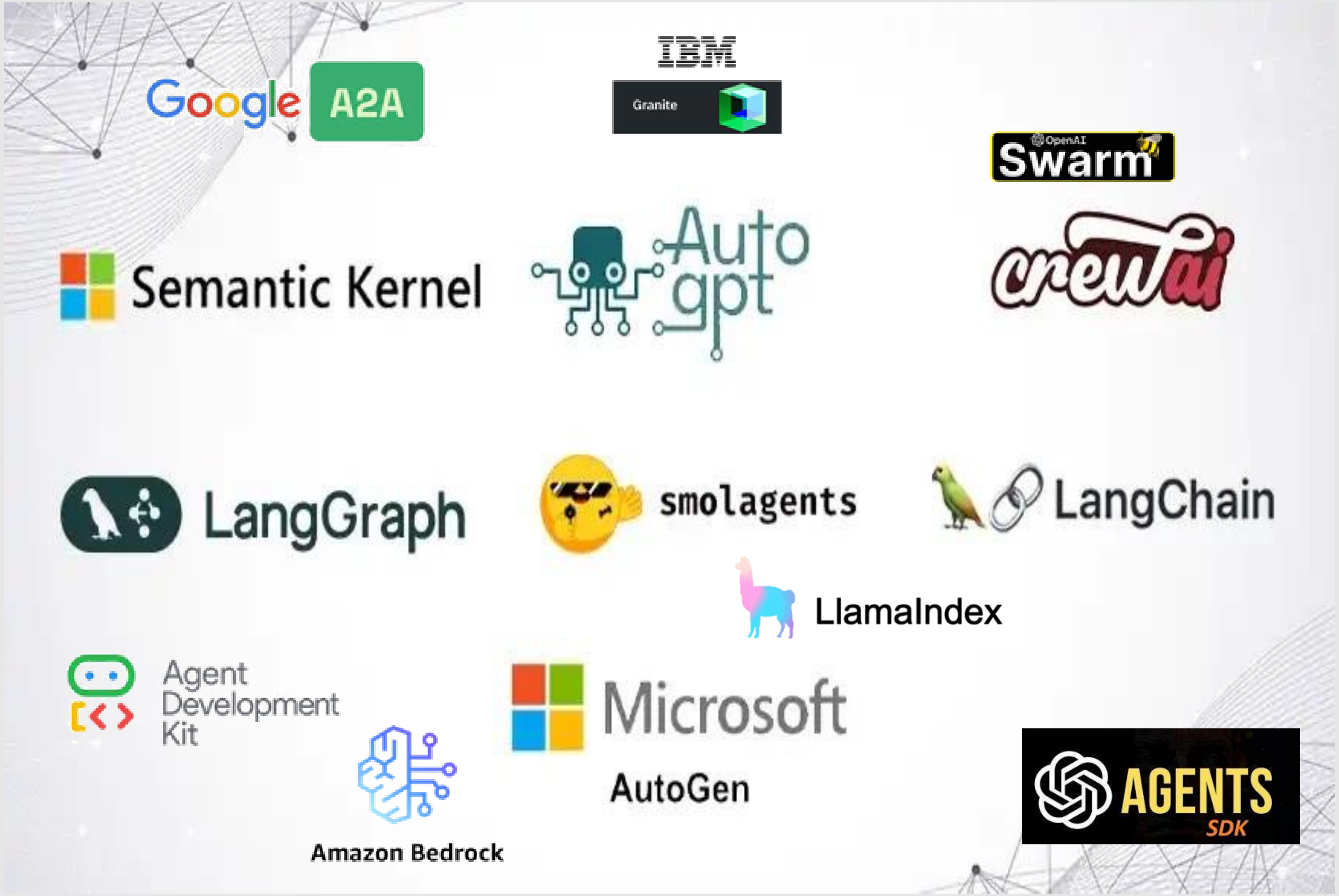
AI Agent Prompting Strategies

Prompting Strategy	Prompting Example	When to Use
Chain-of-Thought (CoT)	'Let's think step by step: If Alice has 3 apples and gives 1 to Bob, how many does she have left?'	Best for logical reasoning, math problems, and structured decision-making.
Tree-of-Thought (ToT)	'Consider multiple possible ways to solve this problem and evaluate each approach before choosing the best one.'	Useful for creative problem-solving, coding strategies, and multi-step reasoning.
Self-Reflection & Self-Critique	'You just provided an answer. Now, evaluate your response and refine it if necessary.'	Great for AI content refinement, debugging, and improving response quality.
ReAct (Reasoning + Acting)	'I need today's weather. First, check a weather API, then summarize the result.'	Ideal for real-time interactions, autonomous AI agents, and API-driven tasks.
Plan-and-Execute	'Before solving this, outline a plan of action. Once the plan is complete, execute each step carefully.'	Best for long-form writing, planning research projects, and structured workflows.
Role-Based Prompting	'You are a cybersecurity analyst. Identify potential vulnerabilities in this system log.'	Great for domain-specific responses such as legal, medical, or technical queries.
Socratic Questioning	'Before answering, ask yourself: What information is missing? What assumptions am I making?'	Useful for AI self-improvement, philosophical debates, and logical consistency.
Debate-Based Prompting	'Present arguments for and against using nuclear energy, then provide a balanced conclusion.'	Best for policy analysis, negotiations, and exploring multiple perspectives.
Incremental Task Completion (Decomposition)	'List the key steps to write an essay on climate change. Complete each step before moving on.'	Ideal for breaking down large tasks, workflow automation, and stepwise execution.
Recursive Criticism & Improvement	'Generate an initial response. Then review it for improvements and provide a refined version.'	Perfect for iterative content refinement, debugging code, and quality enhancement.
Chain-of-Draft (COD)	'Let's think step by step and limit each reasoning step to five words at most.'	Like COT, but uses less tokens and has less latency without a significant drop in accuracy.

Frameworks for Building and Creating AI Agents

- Features:
  - Agent Structure
  - Environment Connection
  - Task Coordination
  - Communication Methods
  - Learning Capabilities
  - System Integration
  - Monitoring & Debugging

- Motivations:
  - **Faster AI Development** Pre-built tools and best practices speed up AI agent creation
  - **Standardized Practices** Encourages consistency, collaboration, and knowledge sharing
  - **Scalability** Supports both simple and complex AI agent systems
  - **Increased Accessibility** Simplifies AI development, making it easier for more people to use
  - **Boosts Innovation** Handles core AI tasks, allowing developers to focus on new advancements

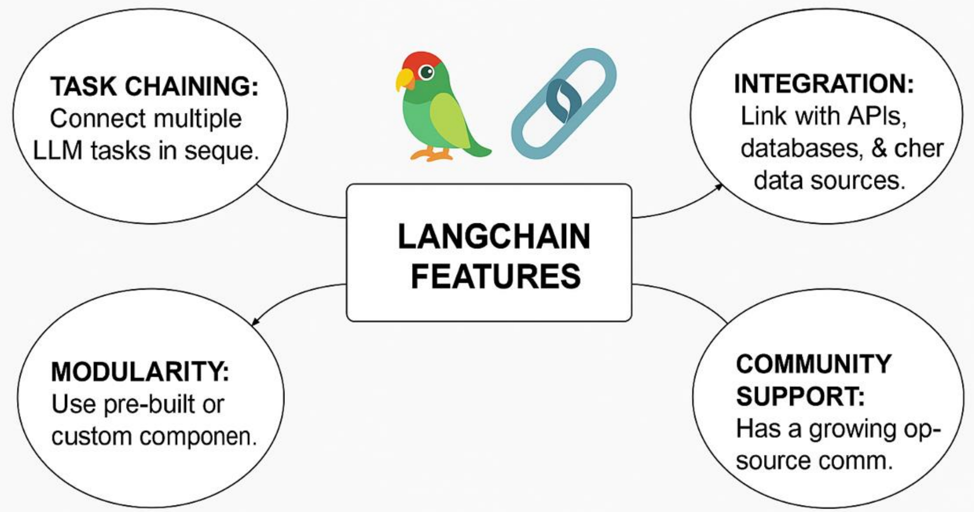


About LangChain

- Modular framework that connects LLMs with tools, data, and memory to build powerful, context-aware apps
- Chains – sequences of modular steps (prompts, tools, LLM calls) that process input and pass output through a defined workflow

- Combines **LLMs with tools**, memory, and control flows
- Supports **agent-based systems** that make decisions dynamically
- Uses **Chains** for step-by-step logic and **Agents** for tool selection
- Enables **integration** with APIs, vector stores, databases, etc.
- Flexible and modular architecture built for LLM application development

Component	Description
Agent	LLM that decides what tools to use and in what order
Chain	A sequence of calls (LLMs, tools, prompts) executed in order
Tool	External function or API an agent can call (e.g., web search, calculator)
Prompt	Templates that structure input for the LLM
Memory	Stores context/history between calls or sessions
Retriever	Interface to search over documents or vector stores



Building AI Agents with LangChain

1. Setting Up a Language Model

```python
from langchain_openai import ChatOpenAI

chat_model = ChatOpenAI(
    api_key="your_api_key",
    model="gpt-4o-mini"
)

response = chat_model.invoke([
    "What is the capital of France?"
])
print(response.content)  # Output: "Paris"
```

2. Defining Tools

```python
from langchain_core.tools import tool

@tool
def save_note(note: str):
    with open("notes.txt", "a") as f:
        f.write(note + "\n")
```

3. Creating an Agent

```python
from langchain.agents import load_tools,
initialize_agent

tools = load_tools(["ddg-search"], llm=llm)

agent = initialize_agent(tools, llm=llm)
agent="zero-shot-react-description", verbose=True)

agent.invoke("What is an AI agent?")
```

4. Executing the Agent

```python
response = agent.invoke({
    "messages": "Save this note: 'Meeting at 3 PM'"
})
print(response["messages"])
```

Lab 1 – Creating a Simple Agent

Purpose: In this lab, we'll learn about the basics of agents and see how tools are called. We'll also see how Chain of Thought prompting works with LLMs and how we can have ReAct agents reason and act..

Architectural Features of AI Agents

Planning

- AI autonomously outlines and executes a logical series of steps for accomplishing a given objective.
- Provides the AI with a way to dynamically adapt its approach based on real-time data and feedback..
- Might employ reflection to evaluate and improve responses
- Example: A research agent plans search → summarize → generate report.

Tool Use

- AI agents interact with external APIs, databases, and functions.
- Enhances LLMs by providing access to real-world knowledge.
- Reduces hallucinations by using retrieval-augmented generation (RAG).
- Example: Calling a Python function to perform complex calculations.

Memory

- Short-term handles tasks; long term stores knowledge and experience
- Memory ensures consistency and efficiency in multi-step decisions
- Memory recalls preferences to enhance personalization and user experience
- Example: Storing user preferences for future reference or personalized responses

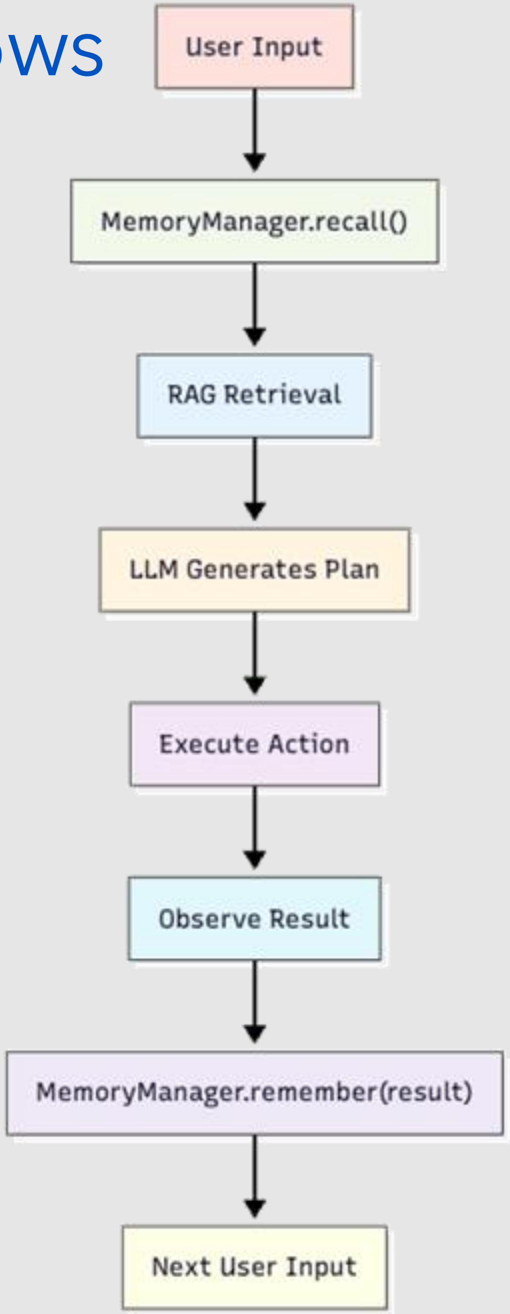
Memory – Why do AI Agents need it?

- **Context Preservation**
  - Retain past interactions to maintain coherent multi-turn dialogue
  - Avoid repetitive questions and redundant processing
- **Personalization & Adaptation**
  - Learn user preferences over time
  - Tailor responses based on historical behavior
- **Efficiency Gains**
  - Cache expensive computations or retrievals
  - Reduce latency by avoiding repeated lookups
- **Robust Decision-Making**
  - Accumulate evidence across steps for more informed planning
  - Support complex reasoning pipelines (e.g., multi-step tasks)

```python
# 1) Context Preservation
history = []
def agent_respond(user_input):
    history.append(user_input)
    context = " ".join(history[-3:])  # last 3 turns
    return llm.generate(context)

# 2) Personalization
user_prefs = {}
def set_pref(key, value):
    user_prefs[key] = value

def greet():
    name = user_prefs.get("name", "there")
    return f"Hi, {name}!"

# 3) Efficiency via Caching
from functools import lru_cache

@lru_cache(maxsize=128)
def expensive_lookup(query: str) -> dict:
    # simulate heavy API call
    return external_api.call(query)

# 4) Robust Decision-Making
evidence = []
def add_evidence(fact):
    evidence.append(fact)

def plan():
    # use accumulated evidence to choose next action
    return planner.decide(evidence)
```

How Memory Integrates into Agent Workflows

- **Retrieval-Augmented Generation (RAG)**
  - Store embeddings of past documents or transcripts
  - Fetch relevant memory snippets during inference
- **Sequential Planning Loops**
  - Write observations and actions into memory after each step
  - Read from memory when deciding next actions
- **Function-Calling Agents**
  - Log previous function calls and results
  - Use historical outputs to inform future tool usage
- **Hierarchical Memory Layers**
  - Short-term cache for immediate turns
  - Mid-term project memory for session-scoped tasks
  - Long-term knowledge base for cross-session continuity



Key Memory Architectures/Backends for Agents

- **Vector Databases (e.g., Chroma, Pinecone)**
  - Store embeddings for similarity search
  - Scale to large memory corpora
- **Knowledge Graphs / Databases**
  - Encode structured facts and relationships
  - Support complex queries over entity networks
- **On-Disk or Cloud Storage**
  - Persist logs, transcripts, and serialized state
  - Enable resumable sessions and audit trails
- **In-Memory Caches**
  - Fast read/write for immediate context
  - Evict stale entries via TTL or LRU policies

```
# Vector DB (Chroma example)
from chromadb import Client
client = Client()
col = client.create_collection("agent_memory")
col.add(documents=["Fact about AI"], embeddings=[embed("Fact about AI")])

# Knowledge Graph (networkx)
import networkx as nx
kg = nx.DiGraph()
kg.add_edge("OpenAI", "GPT-4", relation="developed")

# On-Disk Storage (JSON)
import json
with open("long_term_memory.json", "w") as f:
    json.dump(mem.long_term, f)

# In-Memory Cache (TTL)
from cachetools import TTLCache
cache = TTLCache(maxsize=100, ttl=300)  # entries expire in 5 min
cache["recent_query"] = expensive_lookup("something")
```

smolagents – What is it?

- Lightweight Python agent framework
- Built on Hugging Face ecosystems
- Easy tool-and-memory integration
- Supports synchronous and async execution
- Open-source, minimal dependencies

```
from smolagents import Agent, tool
# Lightweight framework, minimal dependencies

agent = Agent()
# Fast startup, minimal footprint

# Sync execution example
result = agent.run("Hi")
print(result)
# Synchronous execution support
```

smolagents – Core Features

- **Declarative tool registration**
  - Annotate functions with @tool decorator
  - Automatically discovered by agent
- **JSON-schema-driven function calling**
  - Function signature → JSON schema
  - Inputs validated at runtime
- **Pluggable memory backends**
  - In-memory, Redis, or custom stores
  - Swap backend via memory parameter
- **Async/sync agent loops**
  - agent.run() for sync tasks
  - await agent.run_async() for async
- **Built-in final_answer tool**
  - Standardized final output handling
  - Simplifies response formatting

```python
from smolagents import Agent, tool

@tool
# Declarative tool registration
def echo(text: str) -> str:
    """Echo input"""
    return text

agent = Agent(memory="in_memory")
# Pluggable memory backend
agent.register_tool(echo)

# Agent loop example
response = agent.run("echo Hello")
print(response)
```

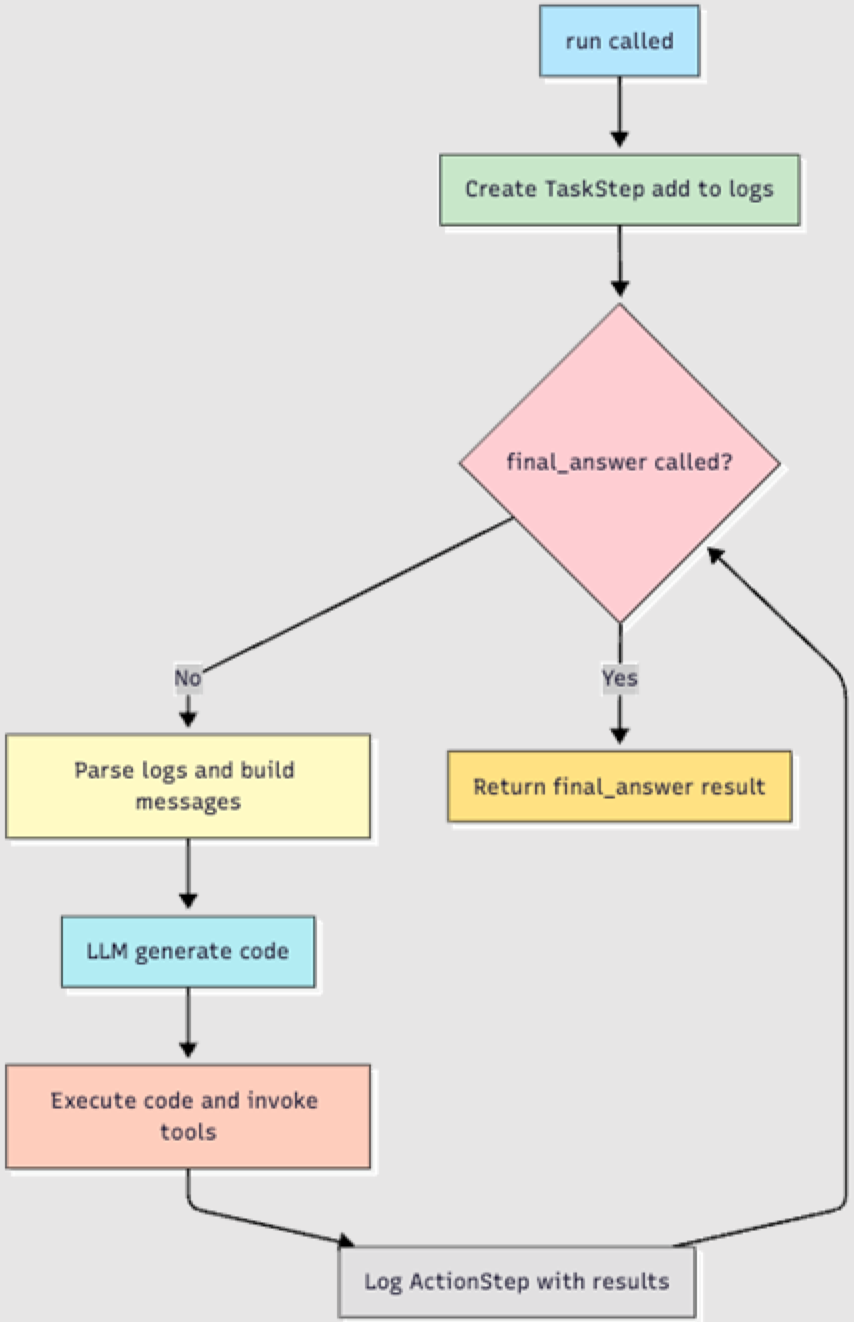
Types of Agents in smolagents

- **ToolCallingAgent** – selects and run registered tools automatically
- **CodeAgent** – generates and executes Python code on-the-fly
- **AsyncAgent** – manages and schedules asynchronous tasks seamlessly
- **ChatAgent** – provides a conversational interface with memory support
- **CustomAgent** – extend the base agent with custom behaviors

```python
from smolagents import (
    ToolCallingAgent, CodeAgent, AsyncAgent, ChatAgent, CustomAgent
)

# ToolCallingAgent: selects and invokes tools
tca = ToolCallingAgent()
print(tca.run("Use weather tool"))    # ← ToolCallingAgent example

# CodeAgent: generates and executes Python code
ca = CodeAgent()
print(ca.run("Write add function"))    # ← CodeAgent example

# AsyncAgent: handles async tasks seamlessly
aa = AsyncAgent()
print(await aa.run_async("Fetch data"))  # ← AsyncAgent example

# ChatAgent: conversational interface
chat = ChatAgent()
print(chat.run("Remember my name"))     # ← ChatAgent example

# CustomAgent: extend base Agent
class MyAgent(CustomAgent):
    pass                                # ← Custom logic entry point
```

Agents That Use Code

- Default CodeAgent writes and executes Python tool calls.
- Uses code over JSON for action specifications.
- Enables composability and reuse of complex operations.
- Manages rich objects (e.g., images, data structures).
- Operates via a ReAct loop with memory logging.
- Define custom tools easily with @tool decorator.
- Runs sandboxed code, with optional authorized imports.
- Supports local or HF API models out of the box.
- Share and load agents via push_to_hub/from_hub.



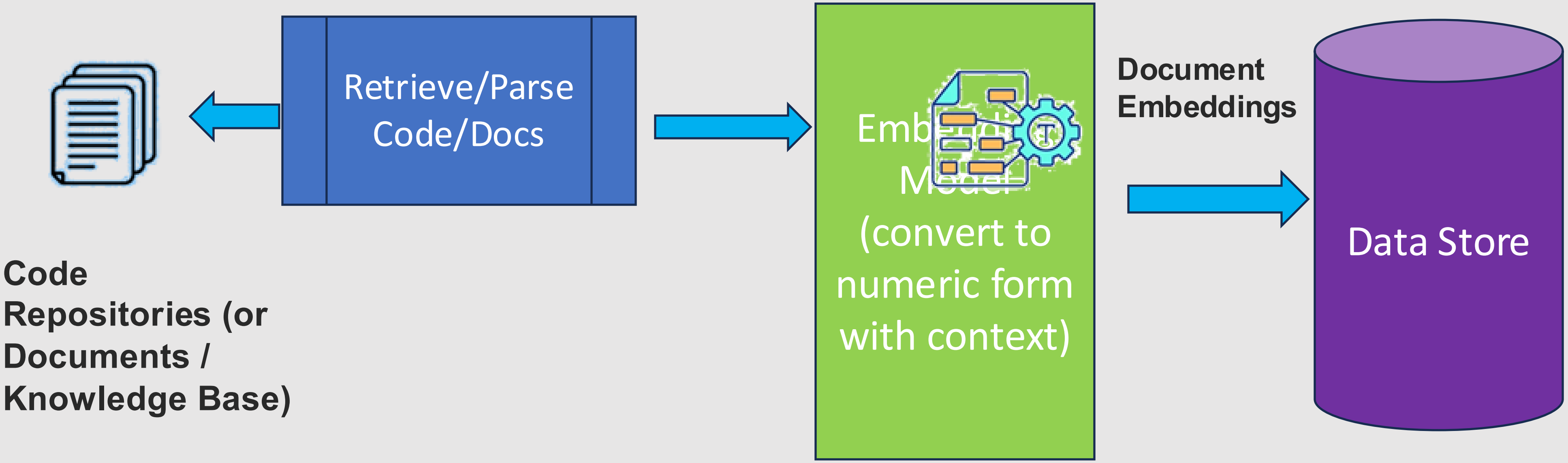
Lab 2 – Leveraging Coding Agents and Persistence

Purpose: In this lab, we'll see how agents can drive solutions via creating code and implement simple memory techniques.

What is Retrieval Augmented Generation (RAG)?

- Search local data for related "hits" and add them to prompt for context
- Unlike keyword search, RAG understands *meaning*
- Your data is turned into numeric representations (embeddings) where each piece of data has information about how it relates to others
- **Retrieval:** When you ask a question/do a search, RAG turns your question/search into its own numeric representation (embedding) and uses calculations to find data that is numerically related (has similar meanings)
- **Augmentation:** The top "hits" (search results) are then added to the prompt we send to the LLM
- **Generation:** Those search results give the LLM some local context (passed in through the prompt) for it to consider in responding

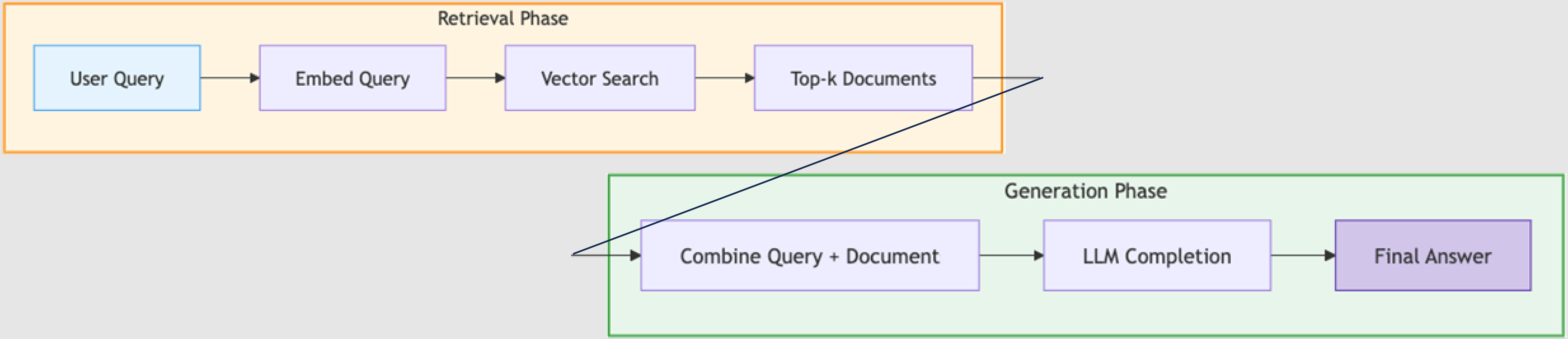
Prepping your data for searching and use with AI



- Your data is parsed and stored with information about other data it's related to in a data store

What is RAG and how does it work? (the application)

- When ready to prompt LLM, a separate "search" is done first to find related information in the data store
  - search strings are parsed and turned into embeddings
  - search is done using calculations on values in vectors to figure out which things are most related
  - top results are returned
- Top results are then added to LLM prompt/query to give it more context to consider
- LLM considers top hits passed to it from your data in addition to its own training data when generating results



Source: https://blogs.nvidia.com/blog/what-is-retrieval-augmented-generation/

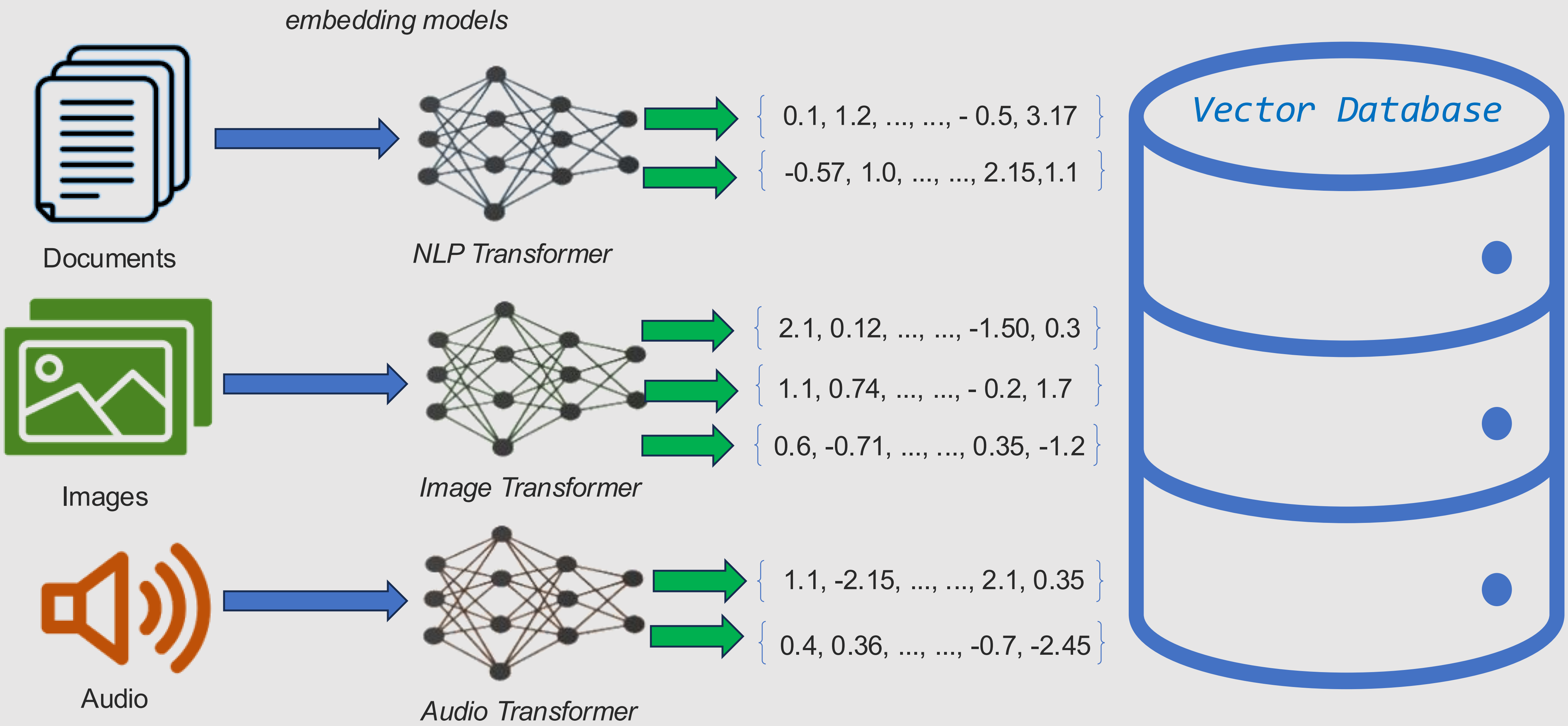
Vector Databases

- Specialized database that index and stores *vector embeddings*
- Useful for
  - fast retrieval
  - similarity search
- Offer comprehensive data management capabilities
  - metadata storage
  - filtering
  - dynamic querying based on associate metadata
- Scalable and can handle large volumes of vector data
- Support real-time updates
- Play key role in AI and ML applications

Vector Database

How data gets into Vector Databases

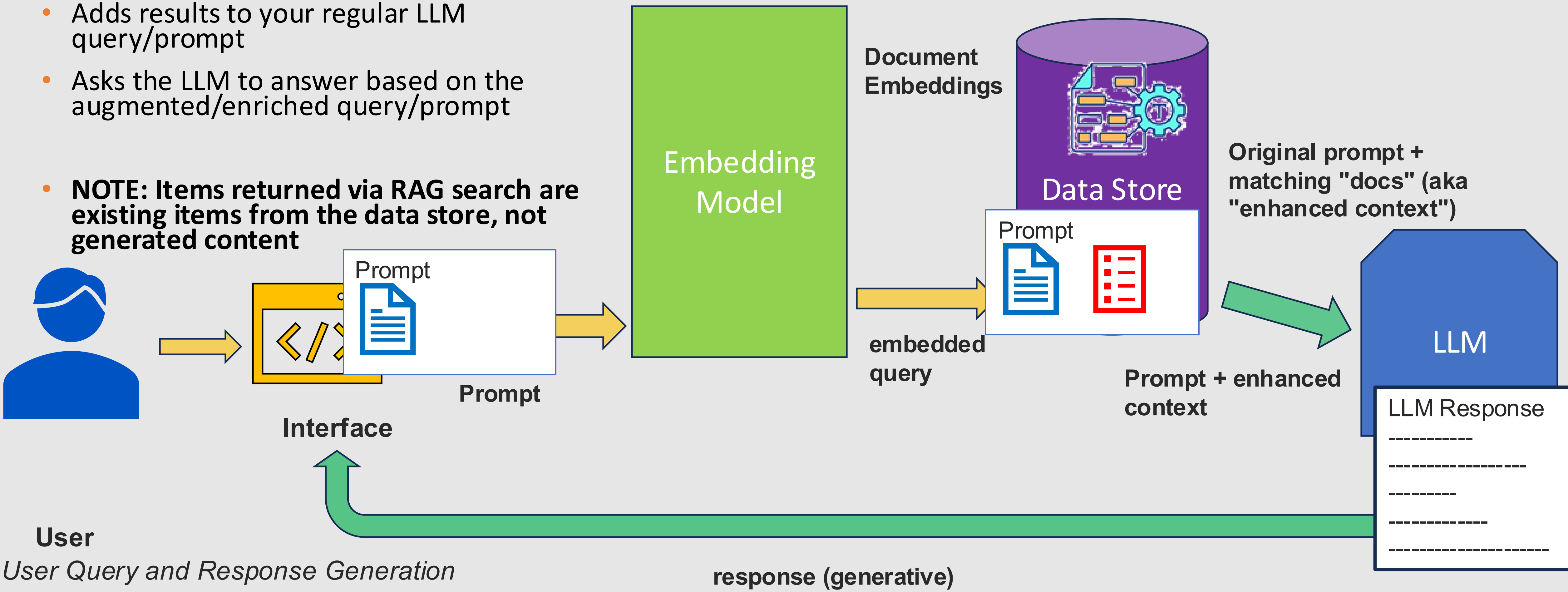
- Data is input, converted to embeddings (vectors) and stored
- Queries are input, converted to embeddings (vectors) and then **similarity metrics** are used to find results ("nearest neighbors")



Integrating your data searches with AI (RAG)

- For queries/prompts, application gathers results (most relevant ones) from the vector database with your data
- Adds results to your regular LLM query/prompt
- Asks the LLM to answer based on the augmented/enriched query/prompt

- **NOTE: Items returned via RAG search are existing items from the data store, not generated content**



What is Agentic RAG?

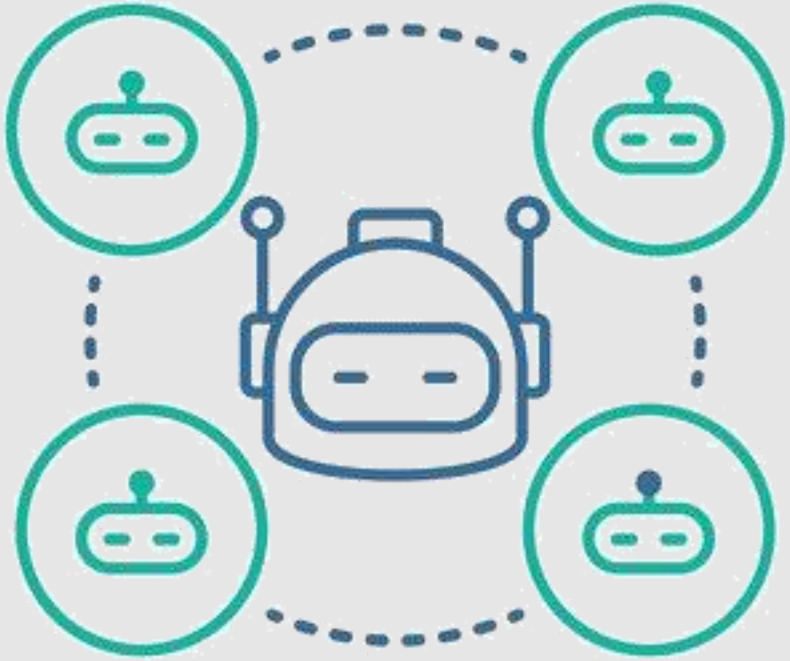
- Integrates AI agents to enhance the RAG approach
- System deconstructs complex queries into manageable parts, processing and using apis/tools where needed to augment processing/get better information
- Agents can
  - analyze original findings from RAG
  - breakdown tasks into subtasks
  - "remember" what steps have been taken and what else needs to be done
  - call an API or tool when needed to solve tasks or get better/more recent info

Lab 3 – Using RAG with Agents

Purpose: In this lab, we'll explore how agents can leverage external data stores via RAG

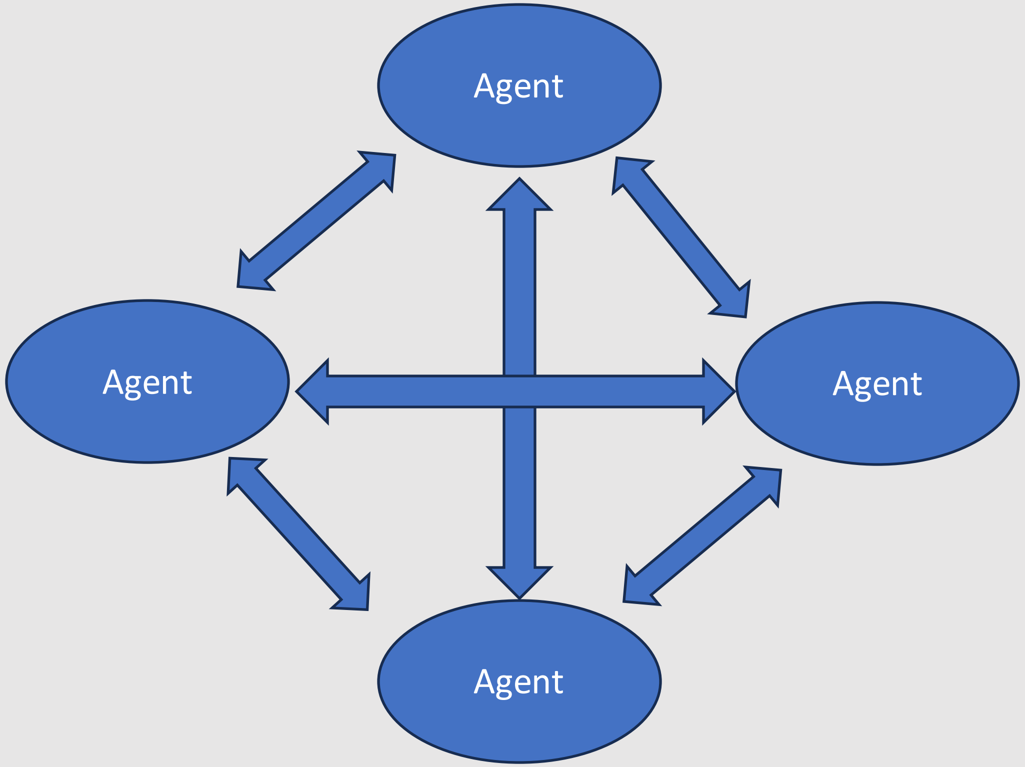
Multi-Agent

- Multiple agents work together to complete tasks efficiently.
- Splits complex tasks into smaller parts and distributes them among specialized agents.
- Uses message passing to coordinate actions and share knowledge.
- Example: A team of agents researches, writes, and fact-checks an article.

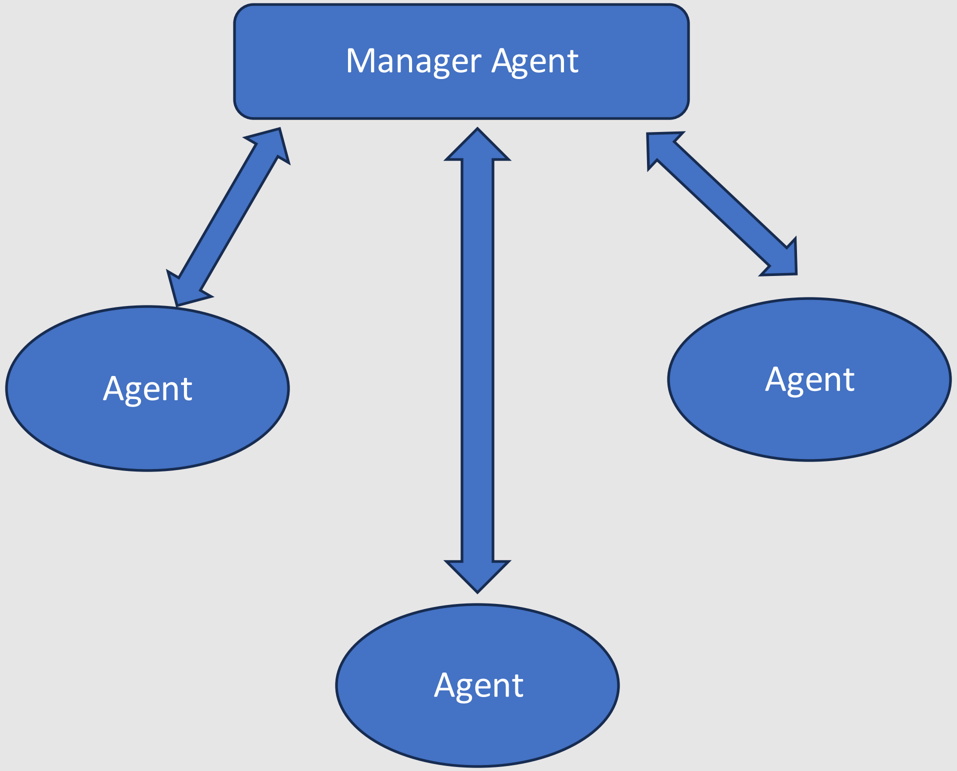


Multi-agent Architectures

Network/Decentralized Architecture



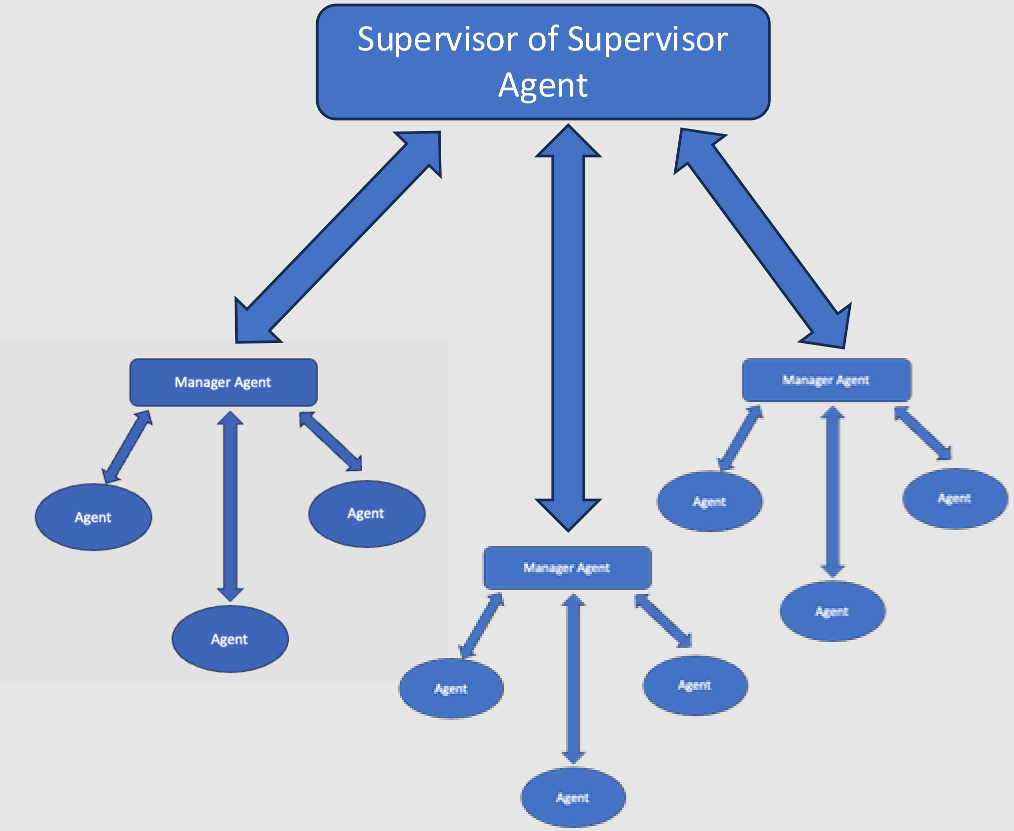
Supervisor/Centralized Architecture



Each agent can communicate with every other agent. Any agent can decide which other agent to call next.

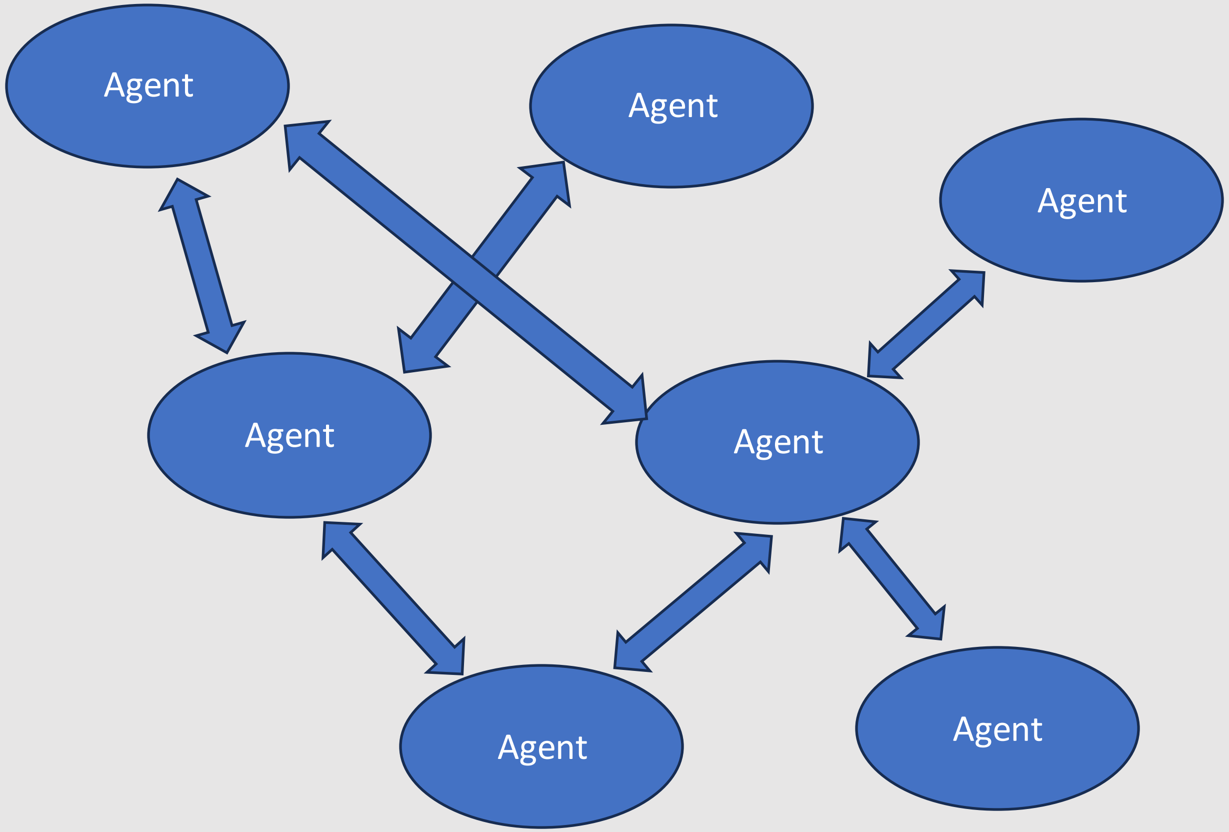
Each agent communicates with a single supervisor agent. Supervisor agent makes decisions on which agent should be called next.

Hierarchical Architecture



Uses a supervisor of supervisors agent. Generalization of supervisor architecture; allows for more complex control flows

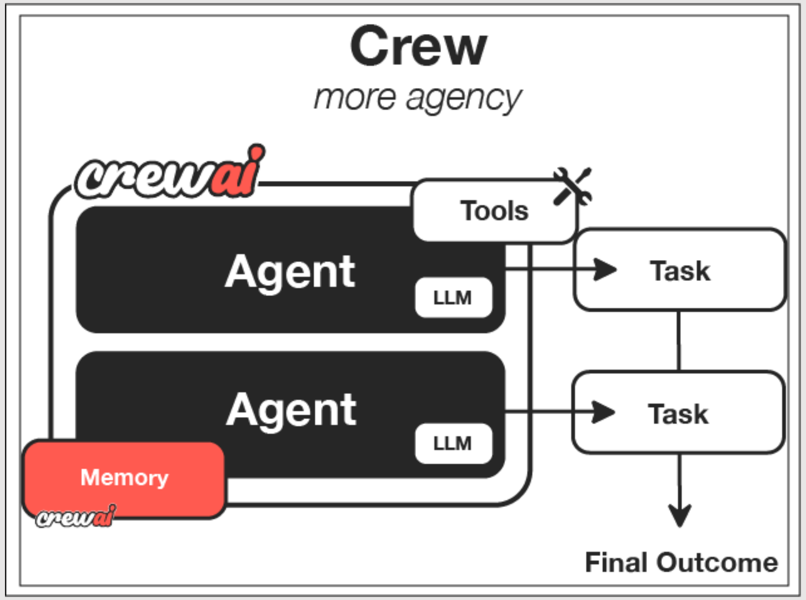
Custom Architecture



Each agent communicates with only a subset of agents; Parts of the flow are deterministic, and only some agents can decide which agents to call next

About CrewAI

Component	Description	Key Features
Crew	The top-level organization	• Manages AI agent teams• Oversees workflows• Ensures collaboration• Delivers outcomes
AI Agents	Specialized team members	• Have specific roles (researcher, writer)• Use designated tools• Can delegate tasks• Make autonomous decisions
Process	Workflow management system	• Defines collaboration patterns• Controls task assignments• Manages interactions• Ensures efficient execution
Tasks	Individual assignments	• Have clear objectives• Use specific tools• Feed into larger process• Produce actionable results



Standard flow

1. Crew organizes overall operation
2. AI agents work on their specialized tasks
3. Process ensures smooth collaboration
4. Tasks get completed to achieve the goal

Source: https://docs.crewai.com/introduction

Building AI Agents with CrewAI

1. Setting Up a Crew

```python
from crewai import Crew, Agent, Task

# Create agents
researcher = Agent(name="Researcher")
writer = Agent(name="Writer")
```

2. Adding a Custom Tool

```python
from crewai.tools import tool

@tool
def search_web(query):
    return f"Results for '{query}'"
researcher.add_tool(search_web)
```

3. Defining Tasks

```python
def summarize_info(topic):
    info = researcher.run_tool(
        "search_web", query=topic
    )
    return writer.ask(
        f"Summarize: {info}"
    )
task = Task(
    description="Summarize topic",
    function=summarize_info,
    args=["AI"]
)
```

4. Running the Crew

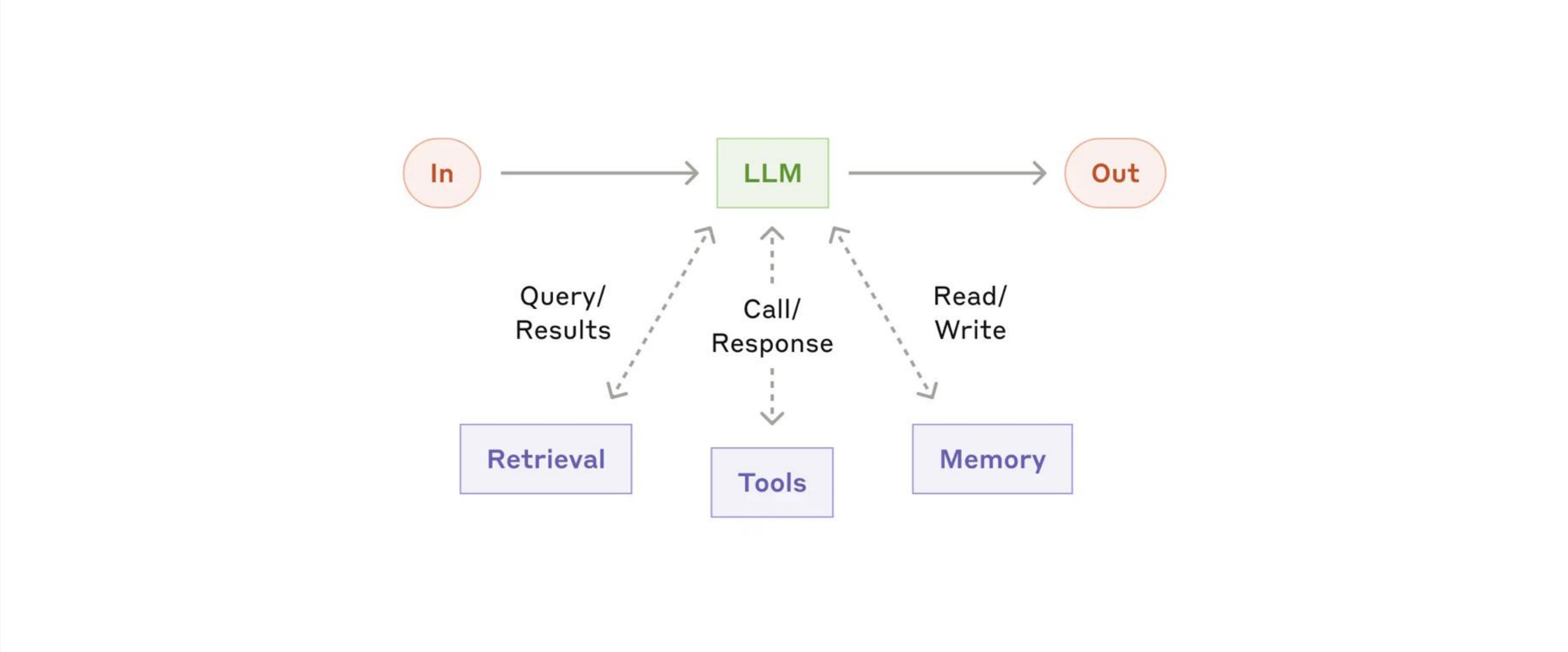
```python
crew = Crew(
    agents=[researcher, writer],
    tasks=[task]
)
result = crew.run()
print(f"Summary: {result}")
```

Lab 4 – Working with Multiple Agents

Purpose: In this lab, we'll see how to add an agent to a workflow using CrewAI.

Agent Design Patterns – Augmented LLM

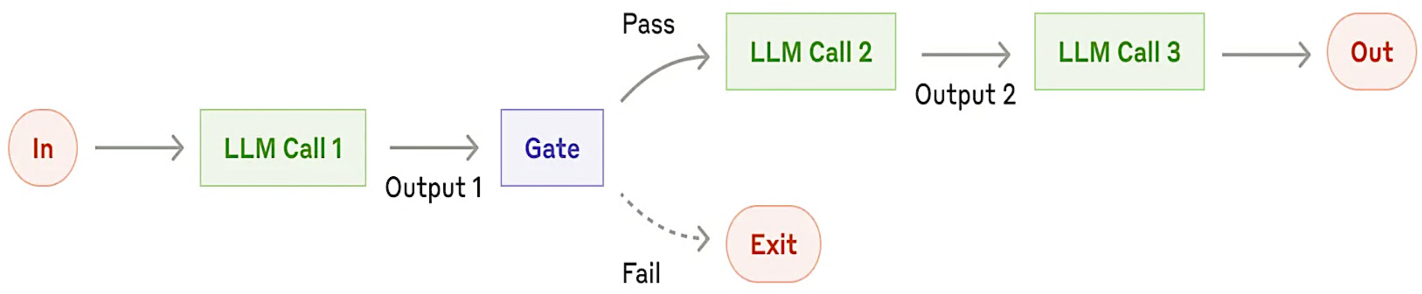
- LLM enhanced with retrieval, tools, memory, etc.
- Models can use these capabilities
  - Generating search queries, selecting tools, deciding what info to keep, etc.



https://www.anthropic.com/research/building-effective-agents

Agent Design Patterns – Workflow: Prompt chaining

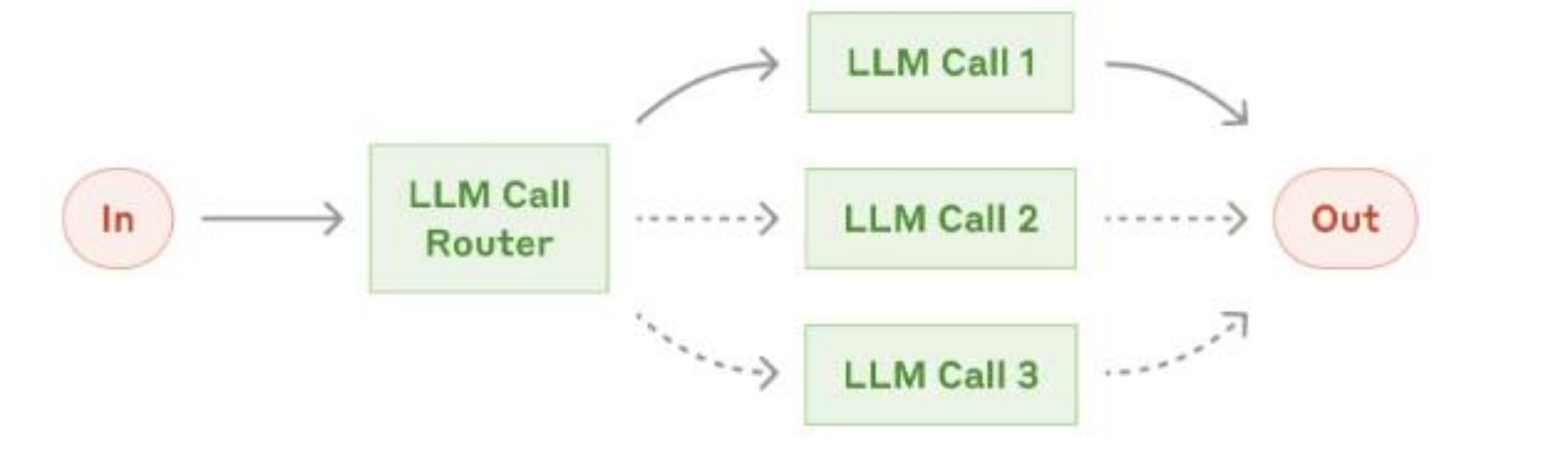
- Decomposes a task into series of steps
- Each LLM call processes output of previous one
- Can add checks ("gates") on any step to make sure things are progressing as expected
- Useful workflow when task can be broken down into fixed subtasks easily and cleanly
- Trades off latency for higher accuracy via each LLM call being a separate task
- Examples:
  - Generating content and translating/editing it
  - Creating an outline and then fleshing it out



https://www.anthropic.com/research/building-effective-agents

Agent Design Patterns – Workflow: Routing

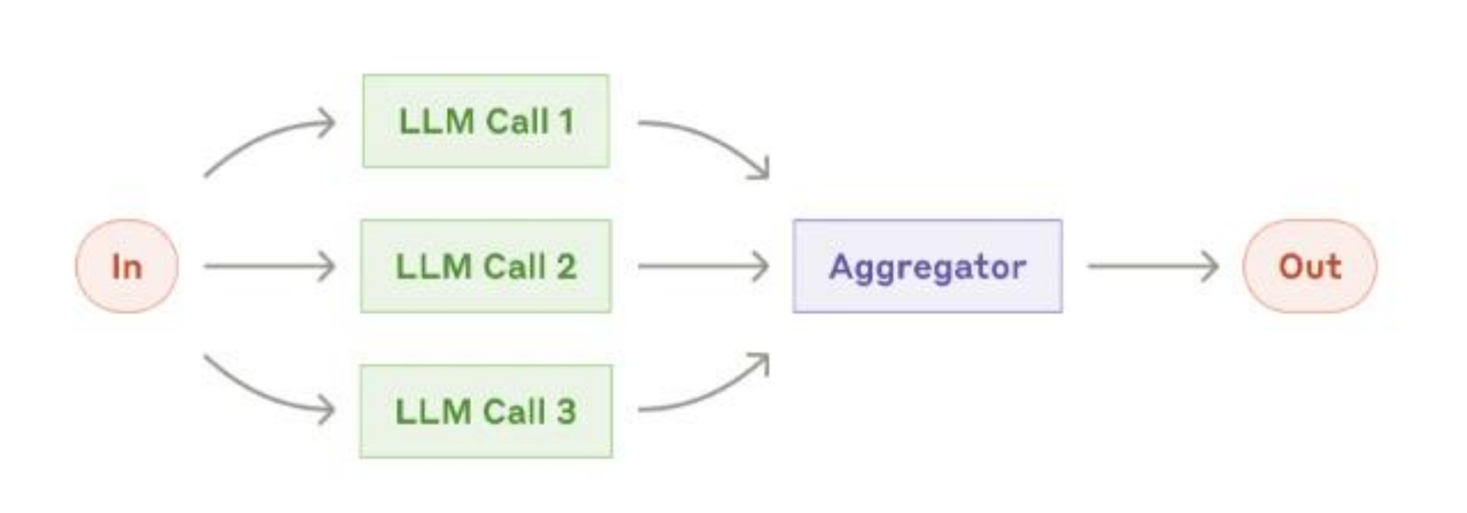
- Classifies input and directs it to specialized follow-up task
- Allows for separation of concerns
- Allows for building specialized prompts
- Useful workflow when task is complex and has different categories that can be handeled separately
- Classification must be accurate
- Examples:
  - Directing different types of customer service queries into different buckets
  - Routing work to smaller models from a larger model to allow larger model to focus on optimization, etc.



https://www.anthropic.com/research/building-effective-agents

Agent Design Patterns – Workflow: Parallelization

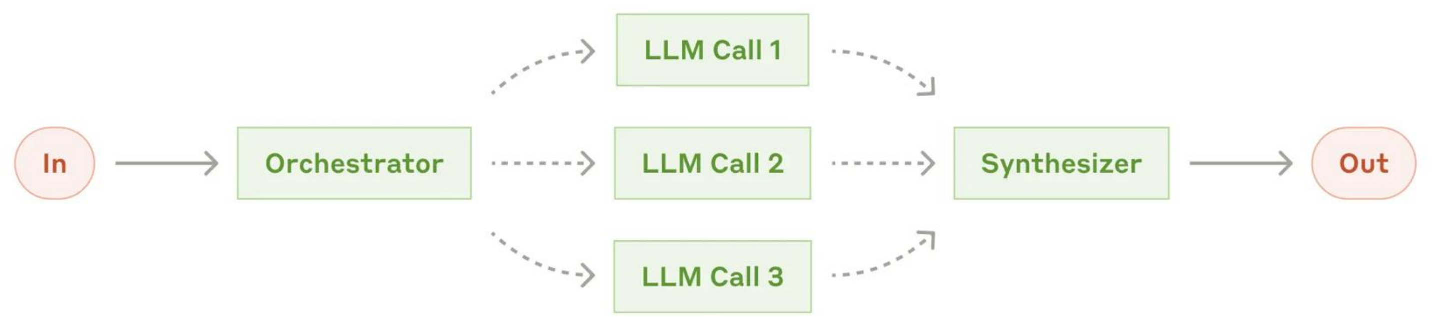
- LLMs work together on a task and have outputs aggregated
- Two variants
  - Sectioning – breaking a task into separate subtasks run in parallel
  - Voting – running the same task multiple times to generate different outputs for comparison
- Useful workflow when
  - Subtasks can be parallelized for speed
  - Multiple "takes" are needed for comparing to get higher confidence
- Examples:
  - Sectioning:
    - Checking user prompts/queries for appropriateness while another instance processes them
  - Voting:
    - Reviewing content (code) for vulnerabilities using multiple different prompts
    - Evaluating content in general with multiple prompts looking at different aspects to come up with net negative or positive



https://www.anthropic.com/research/building-effective-agents

Agent Design Patterns – Workflow: Orchestrator-workers

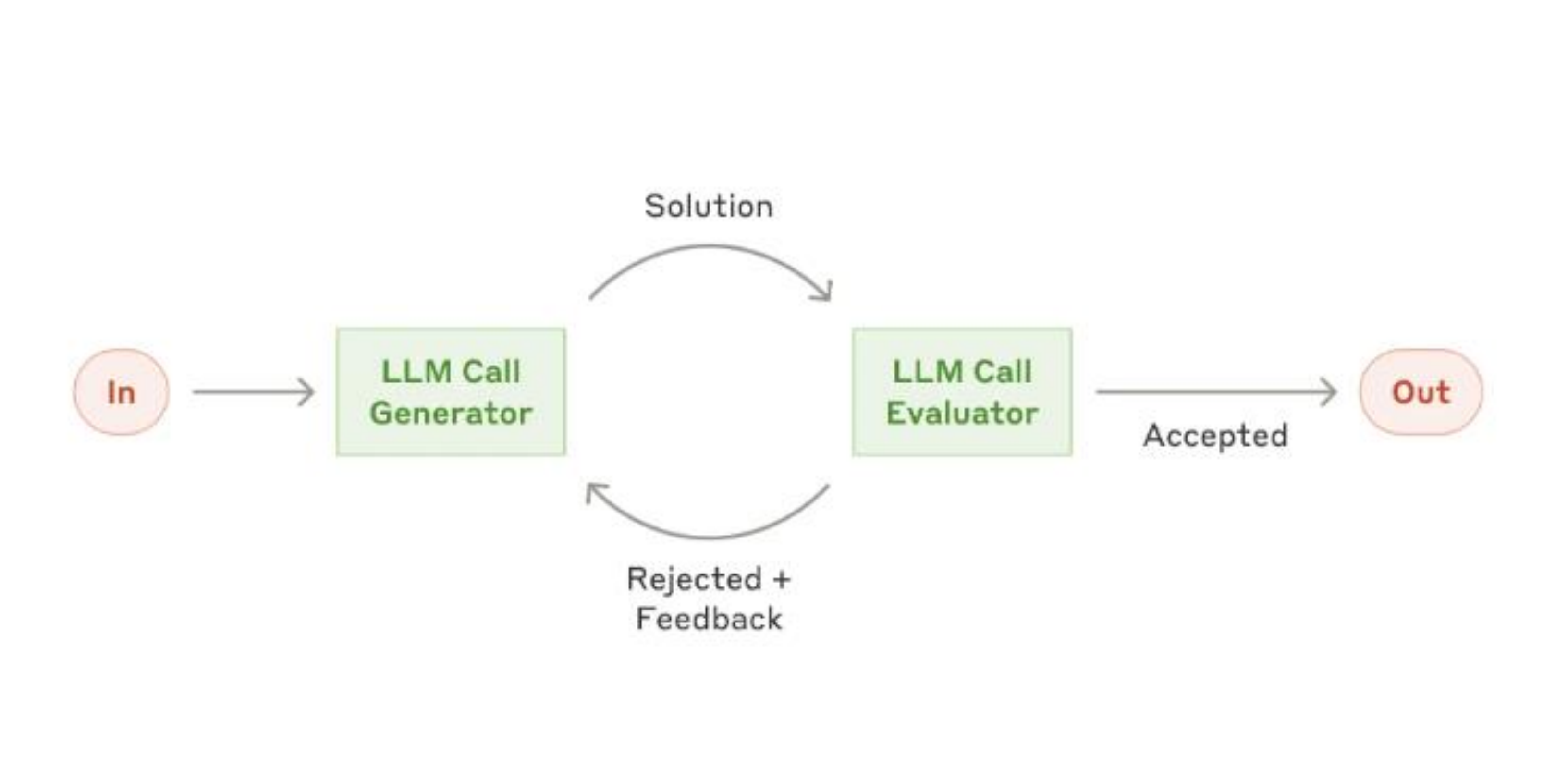
- Central LLM
  - Dynamically splits up tasks
  - Delegates tasks to worker LLMs
  - Synthesizes results
- Useful workflow when
  - Complex task but can't predict subtasks needed
- Difference from parallelization
  - Flexibility – sub-tasks aren't predefined but are determined by the orchestrator
- Examples:
  - Coding tasks that need to make complex changes to multiple files
  - Search tasks that need to gather information from multiple sources



https://www.anthropic.com/research/building-effective-agents

Agent Design Patterns – Workflow: Evaluator-optimizer (aka Reflective)

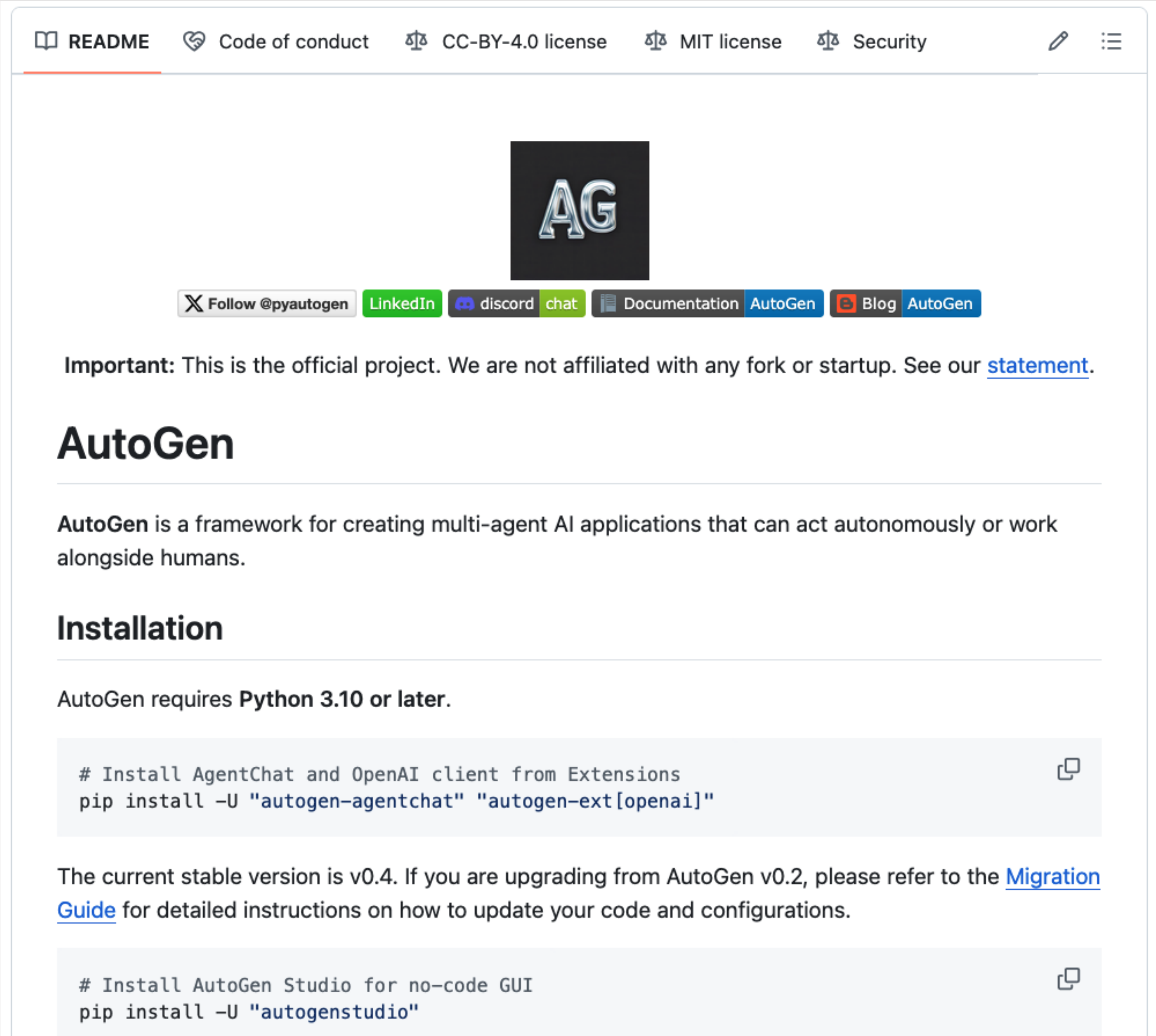
- One LLM call generates a response while another provides evaluation and feedback in loop
- Useful workflow when
  - Clear evaluation criteria
  - When iterative refinement provides value
- Signs of a good fit
  - LLM responses can be improved by a human
  - LLM can provide that kind of feedback
- Examples:
  - Translation
  - Code generation
  - Complex search tasks where evaluator decides if more searches are useful or not



https://www.anthropic.com/research/building-effective-agents

Introduction to AutoGen

- **What is AutoGen?**
  - A Python framework for building multi-agent LLM applications
  - Developed by Microsoft
  - Built for orchestration, tool use, and reasoning
- **Key Design Goals**
  - Enable LLMs to collaborate
  - Support tool calling and reflection
  - Allow flexible agent workflows



README | Code of conduct | CC-BY-4.0 license | MIT license | Security

AG

X Follow @pyautogen | LinkedIn | discord chat | Documentation AutoGen | Blog AutoGen

Important: This is the official project. We are not affiliated with any fork or startup. See our statement.

AutoGen

AutoGen is a framework for creating multi-agent AI applications that can act autonomously or work alongside humans.

Installation

AutoGen requires **Python 3.10 or later.**

```
# Install AgentChat and OpenAI client from Extensions
pip install -U "autogen-agentchat" "autogen-ext[openai]"
```

The current stable version is v0.4. If you are upgrading from AutoGen v0.2, please refer to the Migration Guide for detailed instructions on how to update your code and configurations.

```
# Install AutoGen Studio for no-code GUI
pip install -U "autogenstudio"
```

Why Use AutoGen?

- **Designed for Multi-Agent Workflows**
  - Each agent has a role and memory
  - Agents communicate via messages
- **LLM and Tool Orchestration**
  - Agents can access tools, APIs, and call functions
  - Separation between reasoning and execution
- **Customizable & Extensible**
  - Plug in different models or tool registries
  - Use ReAct-style reasoning patterns

AutoGen Core Concepts

- **Agents**
  - See table
- **Tools**
  - Functions registered via decorators
  - Invoked by LLMs when reasoning leads to tool use
- **Conversations**
  - Agents take turns responding to messages
  - Messages can contain plain text or function calls

Agent Type	Main Role	Key Functions
User Proxy Agent	User interface & task initiation	Starts tasks, relays feedback, enables human-in-the-loop
Assistant Agent	Task execution & content generation	Answers questions, writes code, uses tools
Group Chat Manager	Multi-agent conversation management	Coordinates group chats, manages turn-taking
Function Calling Agent	Executes functions/tools	Calls user-defined functions, integrates with APIs
Custom Agent	User-defined, flexible roles	Combines or extends behaviors for custom workflows

Building AI Agents with AutoGen

1. Setting up Agents

```python
from autogen import UserProxyAgent, AssistantAgent

# Create agents
user = UserProxyAgent(name="user")
assistant = AssistantAgent(name="assistant")
```

2. Adding a Custom Tool

```python
from autogen import register_function

# Define tool function
def search_web(query: str) -> str:
    return f"Results for '{query}'"

# Register tool to assistant
register_function(
    name="search_web",
    description="Search the web for a given query"
)(search_web)
```

3. Defining the Agent Function

```python
# AutoGen uses LLM + tool call during chat
# You do not predefine a task object

# Optional: Provide goal or context in chat
goal = "Find and summarize info about AI"
```

4. Running the Agent

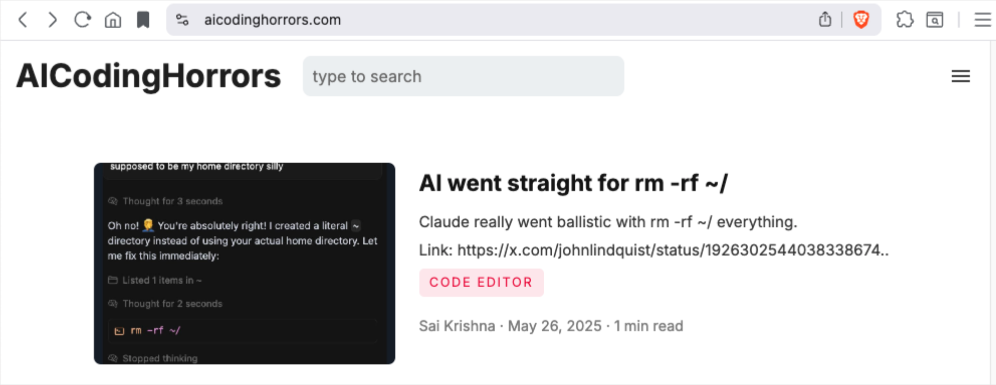
```python
# Start the agent conversation
assistant.initiate_chat(
    user,
    message=goal
)
```

Lab 5 – Building Agents with the Reflective Pattern

Purpose: In this lab, we'll see how to create an agent that uses the reflective pattern using the AutoGen framework.

Too much dependency on LLMs can "break" agents

- Does this need AI?
- Does it need an Agent/LLM?
- Generative ≠ Deterministic
- Frameworks can help...



Tips on building good agents

Simplify workflows

- Reduce LLM calls by grouping tools.
- Use deterministic logic instead of agentic decisions.

Use deterministic logic

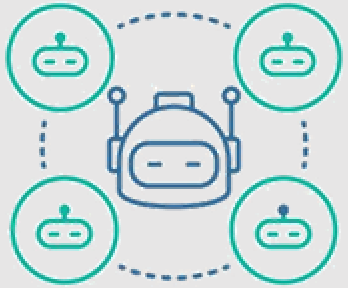
- Minimize reliance on LLM decisions.
- Implement clear, predictable functions.

Improve tool logging

- Log all tool execution details.
- Include error details for better debugging.

Clarify task formulations

- Ensure tasks are clearly defined.
- Provide detailed context for the LLM.



Use stronger LLMs

- Upgrade to more powerful models.
- Avoid errors due to weak reasoning.

Provide extra guidance

- Add task details for clarity.
- Enhance tool descriptions for better use.

Change system prompts carefully

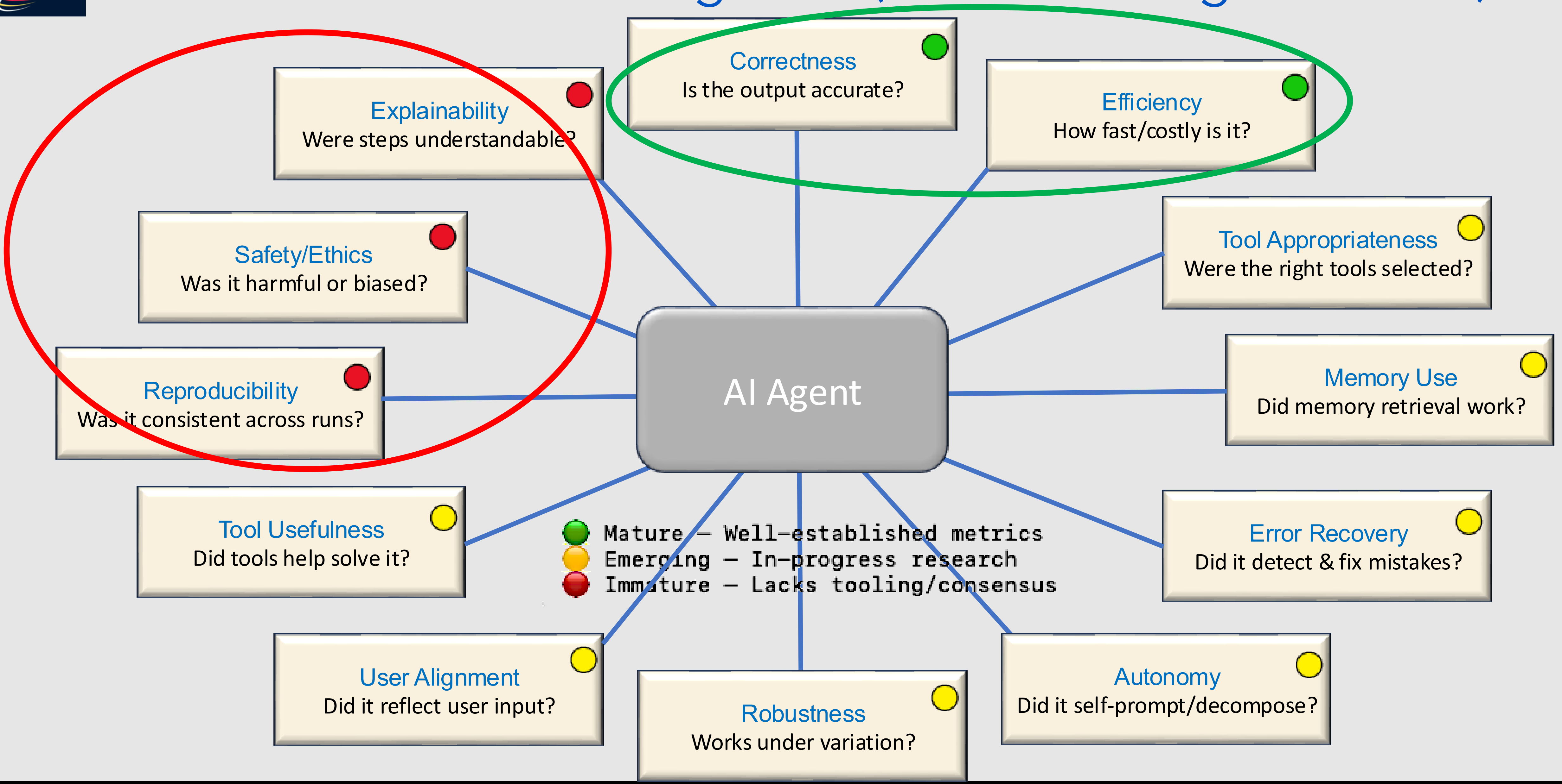
- Avoid altering default prompts unless necessary.
- Ensure custom prompts include required placeholders.

Plan regularly

- Implement regular planning steps.
- Reflect on known facts to guide actions.

Adapted from: https://huggingface.co/docs/smolagents/main/en/tutorials/building_good_agents

How Do We Measure Agents? (AKA Challenges & Risks)



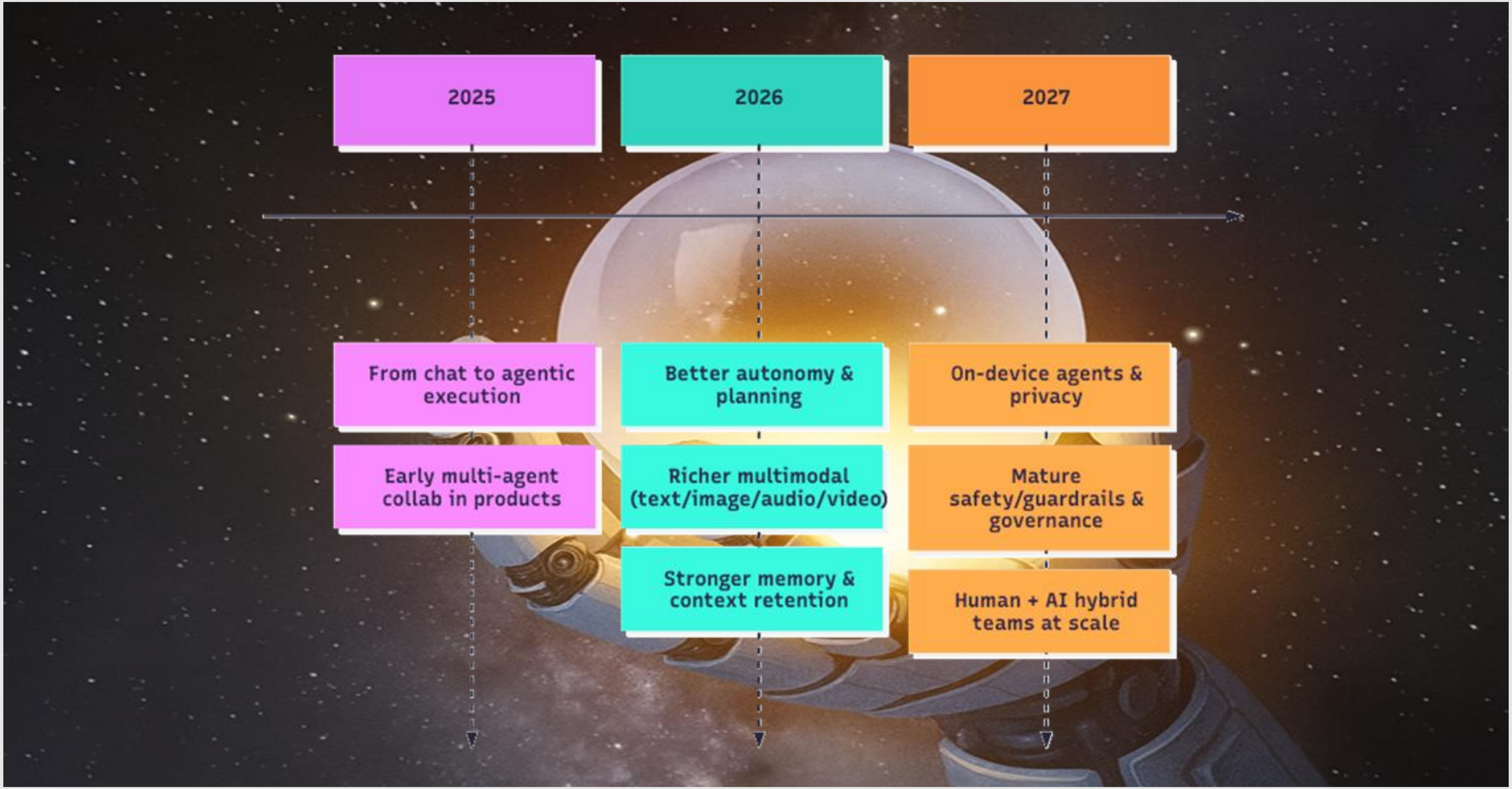
The good news… key areas are maturing (safety & alignment)

Key takeaway: ***The future of AI agents continues to look promising, but needs time and work to mature…***

🤖 AI Agent

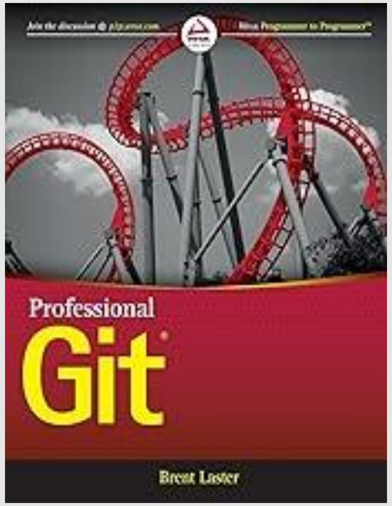
Example:
If a tool tries to run a shell command, the sandbox blocks it and the anomaly detector raises an alert.

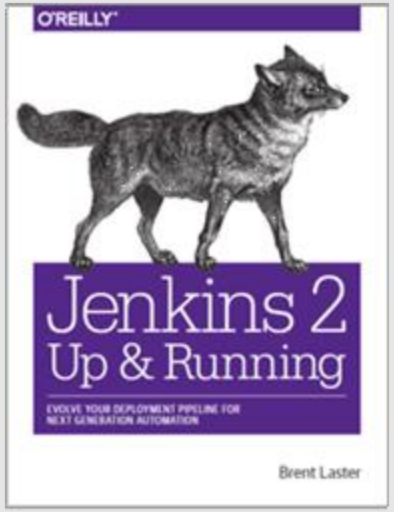
Future of AI Agents

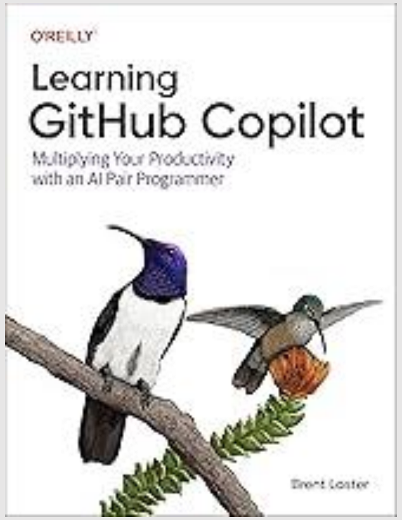


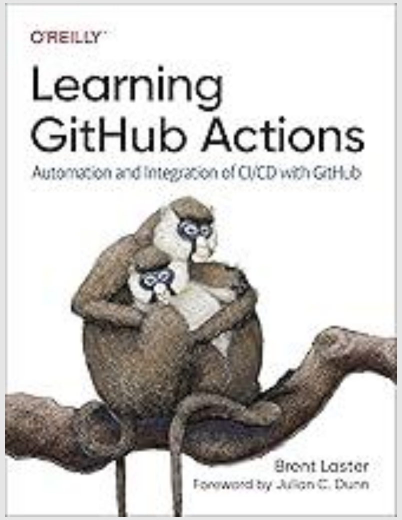
That's all - thanks!

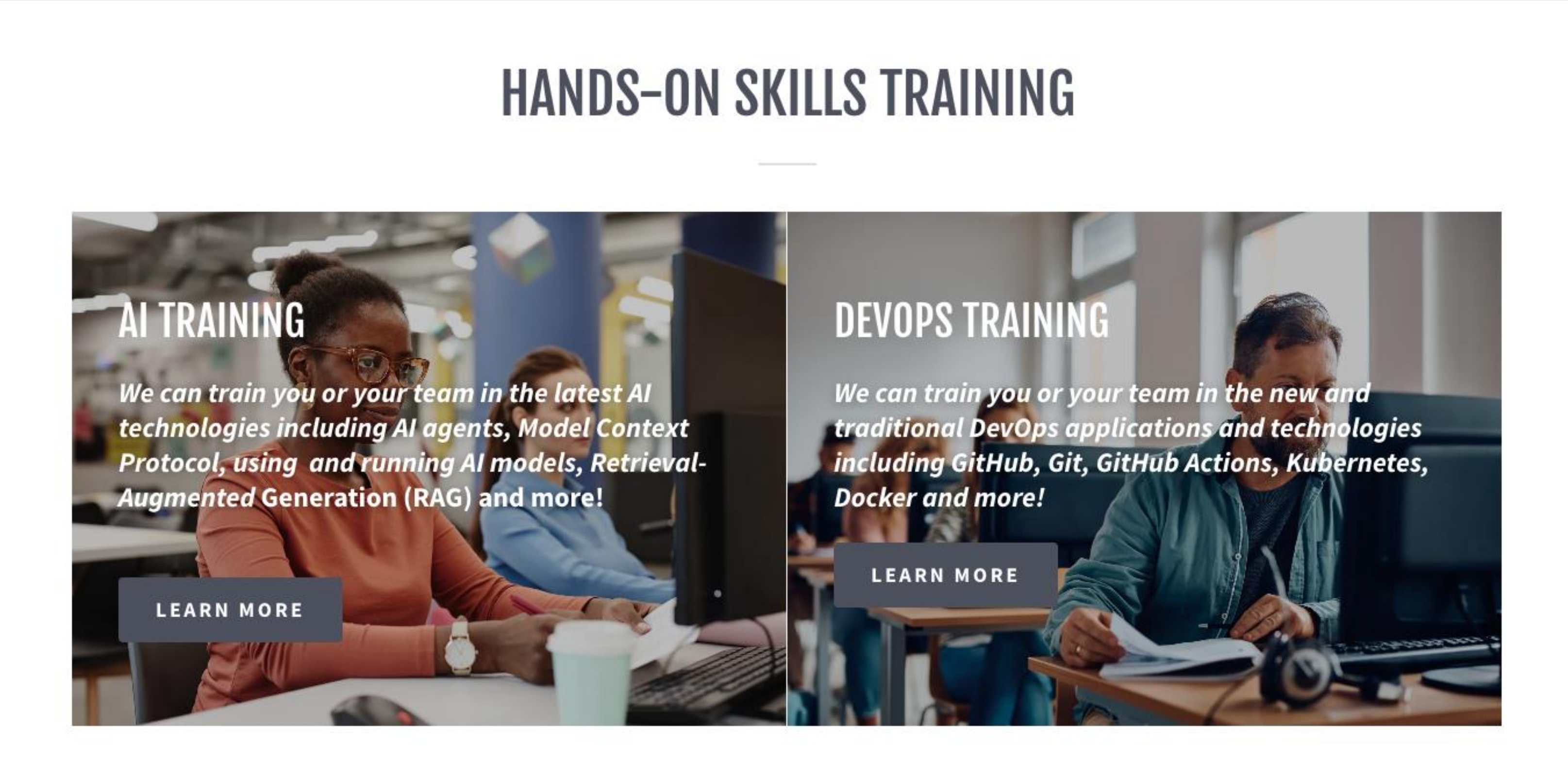
Contact: training@getskillsnow.com











techskillstransformations.com
getskillsnow.com